

BOOK 4: INFRASTRUCTURE AND SCALING

BLOCKCHAIN PRODUCTION SYSTEMS

INTRODUCTION

Building a blockchain application that works on your laptop is one thing. Running it for thousands of users with 99.9% uptime is entirely different. This book bridges that gap between development and production, teaching you the infrastructure patterns that separate hobby projects from professional systems.

The blockchain space has a dirty secret. Most tutorials end when the smart contract deploys. They never mention that your application will fail when RPC rate limits hit. They skip the part where your database fills with millions of events. They ignore the reality that users expect instant responses while blockchain confirmations take minutes.

This book fills those gaps with battle-tested infrastructure patterns.

WHAT THIS BOOK COVERS

Chapter 1: RPC Node Infrastructure

Understanding the backbone of all blockchain interactions. Running your own nodes versus using providers. Load balancing across multiple endpoints. Handling rate limits, timeouts, and failovers. Choosing between full nodes, archive nodes, and light clients.

Chapter 2: Event Indexing and Data Pipelines

Blockchain data is not queryable by default. Building indexers that track every event. Database schemas optimized for blockchain data. Real-time streaming versus batch processing. Handling chain reorganizations without corrupting your data.

Chapter 3: Caching Strategies

Users expect millisecond responses but blockchains operate on

seconds to minutes. Multi-layer caching from memory to Redis to database. Cache invalidation strategies for blockchain data. Balancing freshness against performance.

Chapter 4: Transaction Management

Sending transactions reliably in production. Nonce management across multiple services. Gas price strategies and stuck transaction recovery. Transaction queuing and prioritization. Handling the unpredictable mempool.

Chapter 5: Monitoring and Alerting

You cannot fix what you cannot see. Metrics collection for blockchain applications. Log aggregation and structured logging. Alerting on anomalies before users notice. Dashboards that actually help during incidents.

Chapter 6: High Availability Architecture

Designing systems that survive failures. Database replication and failover. Stateless service design. Geographic distribution for global users. Disaster recovery planning for blockchain applications.

Chapter 7: Security in Production

Protecting private keys at scale. Hardware security modules and key management services. API authentication and rate limiting. DDoS protection for blockchain services. Audit logging and compliance.

Chapter 8: Cost Optimization

Blockchain infrastructure is expensive. Optimizing RPC costs through batching and caching. Database storage strategies. Compute right-sizing. Understanding the true cost of running blockchain services.

THE PRODUCTION MINDSET

Development asks whether something works. Production asks whether it keeps working at 3 AM on a holiday weekend when traffic spikes 10x and your third-party provider is having issues.

Production systems need defense in depth. Every external

dependency will fail eventually. Your RPC provider will have outages. Your database will run out of connections. Your cache will evict data at the worst possible time. The systems in this book anticipate these failures and handle them gracefully.

WHO THIS BOOK IS FOR

Developers ready to move beyond localhost. Engineers tasked with making blockchain applications reliable. DevOps professionals entering the blockchain space. Technical leads responsible for system architecture.

You should already understand blockchain basics, smart contracts, and application development from the previous books. This book assumes you can write the code and focuses on running it reliably.

THE INFRASTRUCTURE STACK

Throughout this book, we build a complete infrastructure stack:

The data layer handles blockchain connectivity with multi-provider RPC management, event indexing with PostgreSQL and Elasticsearch, caching with Redis, and time-series metrics with InfluxDB or Prometheus.

The application layer includes transaction management services, API gateways with rate limiting, background job processing, and WebSocket servers for real-time updates.

The operations layer covers monitoring with Grafana dashboards, alerting with PagerDuty integration, log aggregation with the ELK stack, and deployment automation with container orchestration.

Each chapter builds on this stack, adding components and patterns until you have a production-grade infrastructure.

TECHNOLOGY CHOICES

This book uses specific technologies but teaches transferable patterns:

PostgreSQL for relational data because it handles blockchain data well with its JSON support and partitioning capabilities.

Redis for caching because it is the industry standard with excellent performance characteristics.

Docker and Kubernetes for deployment because container orchestration is essential for modern infrastructure.

Prometheus and Grafana for monitoring because they are open source, powerful, and widely adopted.

The patterns matter more than the specific tools. If your organization uses different technologies, the concepts transfer directly.

COST REALITY

Running blockchain infrastructure costs real money. A single archive node can cost hundreds per month. RPC providers charge per request. Indexing the full history of a chain requires terabytes of storage.

This book does not pretend infrastructure is free. Each chapter discusses cost implications and optimization strategies. You will learn to make informed tradeoffs between performance, reliability, and budget.

SCALING PHILOSOPHY

Scaling is not about handling more traffic. Scaling is about maintaining quality of service as demand increases. A system that handles 1000 requests per second poorly is not scaled. A system that handles 100 requests per second with consistent latency and

reliability is well-scaled for its load.

We focus on efficiency before horizontal scaling. Optimize what you have before adding more instances. Many blockchain applications never need to scale beyond a single well-tuned server. Others need global distribution across dozens of nodes. This book prepares you for both scenarios.

STARTING POINT

Before diving in, you should have:

A development environment with Docker installed. Familiarity with at least one cloud provider. Basic understanding of databases and SQL. Comfort with the command line.

The code examples use Python and JavaScript as primary languages, with configuration in YAML and shell scripts for automation. All examples are production-grade, not simplified demos.

LET US BUILD INFRASTRUCTURE THAT LASTS

The blockchain space is littered with projects that launched successfully but collapsed under real-world load. The teams behind those failures knew how to write smart contracts but not how to run reliable services.

This book ensures you are not among them. By the end, you will have the knowledge to run blockchain applications that stay up, stay fast, and scale with demand.

The infrastructure starts now.

CHAPTER 1: RPC NODE INFRASTRUCTURE

Every blockchain application communicates with the network through RPC nodes. These nodes are the gateway between your code and the blockchain. This chapter covers everything about

running, connecting to, and managing RPC infrastructure for production systems.

SECTION 1: UNDERSTANDING RPC NODES

RPC stands for Remote Procedure Call. An RPC node is a blockchain node that exposes an API for external applications to read data and submit transactions. Without RPC nodes, your application has no way to interact with the blockchain.

There are three types of nodes you need to understand.

Full nodes store the current state of the blockchain. They can answer queries about current balances, execute calls against current contracts, and validate new transactions. Full nodes typically store several hundred gigabytes of data and can respond to most application queries.

Archive nodes store the complete history of the blockchain including every historical state. They can answer queries about any block in history such as what was address X's balance at block 1000000. Archive nodes require multiple terabytes of storage and are significantly more expensive to run.

Light clients do not store blockchain data locally. They query full nodes for the data they need and verify it cryptographically. Light clients are useful for mobile applications or environments with limited resources but cannot serve as RPC endpoints for other applications.

The choice between these affects your infrastructure. Most applications need full node access for current data. Applications that analyze historical data or implement complex queries need archive node access.

SECTION 2: RPC PROVIDERS VERSUS SELF-HOSTED

You have two options for RPC access. Use a provider like Alchemy,

Infura, or QuickNode. Or run your own nodes.

RPC providers handle the infrastructure complexity. They run redundant nodes across multiple data centers, handle synchronization, manage updates, and provide high availability. You pay per request or for a monthly quota.

Here is how to configure multiple providers for redundancy.

```
import os
import time
import random
from typing import Optional, Dict, Any, List
from dataclasses import dataclass
from enum import Enum
import httpx
import asyncio

class ProviderStatus(Enum):
    HEALTHY = "healthy"
    DEGRADED = "degraded"
    DOWN = "down"

@dataclass
class RPCProvider:
    name: str
    url: str
    priority: int
    weight: int
    max_requests_per_second: int
    status: ProviderStatus = ProviderStatus.HEALTHY
    consecutive_failures: int = 0
    last_request_time: float = 0
    request_count: int = 0

class RPCProviderManager:
    def __init__(self, providers: List[Dict[str, Any]]):
        self.providers = []
        for config in providers:
            provider = RPCProvider(
```

```
name = config["name"],
url = config["url"],
priority = config.get("priority", 1),
weight = config.get("weight", 1),
max_requests_per_second = config.get("rps_limit", 25)
)
self.providers.append(provider)
```

```
self.providers.sort(key = lambda p: p.priority)
self.failure_threshold = 3
self.recovery_time = 60
self.last_health_check = 0
```

```
def get_available_providers(self) -> List[RPCProvider]:
    available = []
    current_time = time.time()
```

```
    for provider in self.providers:
        if provider.status == ProviderStatus.DOWN:
            if current_time - provider.last_request_time > self.recovery_time:
                provider.status = ProviderStatus.DEGRADED
                provider.consecutive_failures = 0
            else:
                continue
```

```
    time_since_last = current_time - provider.last_request_time
    if time_since_last >= 1:
        provider.request_count = 0
```

```
    if provider.request_count < provider.max_requests_per_second:
        available.append(provider)
```

```
    return available
```

```
def select_provider(self) -> Optional[RPCProvider]:
    available = self.get_available_providers()
```

```
    if not available:
        return None
```



```
healthy = [p for p in available if p.status ==
ProviderStatus.HEALTHY]
if healthy:
    available = healthy
```

```
total_weight = sum(p.weight for p in available)
choice = random.uniform(0, total_weight)
```

```
current = 0
for provider in available:
    current += provider.weight
if choice <= current:
    return provider
```

```
return available[0]
```

```
def record_success(self, provider: RPCProvider):
    provider.consecutive_failures = 0
    provider.request_count += 1
    provider.last_request_time = time.time()
```

```
if provider.status == ProviderStatus.DEGRADED:
    provider.status = ProviderStatus.HEALTHY
```

```
def record_failure(self, provider: RPCProvider):
    provider.consecutive_failures += 1
    provider.request_count += 1
    provider.last_request_time = time.time()
```

```
if provider.consecutive_failures >= self.failure_threshold:
    provider.status = ProviderStatus.DOWN
elif provider.consecutive_failures >= 1:
    provider.status = ProviderStatus.DEGRADED
```

```
def get_stats(self) -> Dict[str, Any]:
    return {
        "providers": [
            {
                "name": p.name,
                "status": p.status.value,
```

```

"failures": p.consecutive_failures,
"requests": p.request_count
}
for p in self.providers
]
}

```

The provider manager tracks health status, respects rate limits, and distributes load across multiple providers. When a provider fails repeatedly, it gets marked as down and traffic shifts to healthy providers.

SECTION 3: BUILDING A RESILIENT RPC CLIENT

A production RPC client needs retry logic, timeout handling, and automatic failover. Here is a complete implementation.

```

class ResilientRPCClient:
    def __init__(self, provider_configs: List[Dict[str, Any]]):
        self.manager = RPCProviderManager(provider_configs)
        self.client = httpx.AsyncClient(timeout = 30.0)
        self.request_id = 0
        self.max_retries = 3
        self.retry_delay = 1.0

```

```

    async def call(
        self,
        method: str,
        params: List[Any] = None,
        timeout: float = None
    ) -> Any:

```

```

        params = params or []
        last_error = None

```

```

        for attempt in range(self.max_retries):
            provider = self.manager.select_provider()

```

```

            if not provider:

```

```
await asyncio.sleep(self.retry_delay * (2 ** attempt))
continue
```

```
try:
    result = await self._make_request(provider, method, params,
    timeout)
    self.manager.record_success(provider)
    return result
```

```
except RPCError as e:
    self.manager.record_failure(provider)
    last_error = e
```

```
if not e.retryable:
    raise
```

```
await asyncio.sleep(self.retry_delay * (2 ** attempt))
```

```
except Exception as e:
    self.manager.record_failure(provider)
    last_error = e
    await asyncio.sleep(self.retry_delay * (2 ** attempt))
```

```
raise RPCConnectionError(f'All providers failed after
{self.max_retries} attempts: {last_error}')
```

```
async def _make_request(
    self,
    provider: RPCProvider,
    method: str,
    params: List[Any],
    timeout: float = None
) -> Any:
```

```
self.request_id += 1
```

```
payload = {
    "jsonrpc": "2.0",
    "id": self.request_id,
    "method": method,
```

```
"params": params
}
```

```
request_timeout = timeout or 30.0
```

```
try:
```

```
    response = await self.client.post(
        provider.url,
        json=payload,
        timeout=request_timeout
    )
```

```
    response.raise_for_status()
```

```
except httpx.TimeoutException:
```

```
    raise RPCError("Request timeout", retryable=True)
```

```
except httpx.HTTPStatusError as e:
```

```
    if e.response.status_code == 429:
```

```
        raise RPCError("Rate limited", retryable=True)
```

```
    elif e.response.status_code >= 500:
```

```
        raise RPCError(f"Server error: {e.response.status_code}",
            retryable=True)
```

```
    else:
```

```
        raise RPCError(f"HTTP error: {e.response.status_code}",
            retryable=False)
```

```
data = response.json()
```

```
if "error" in data:
```

```
    error = data["error"]
```

```
    code = error.get("code", 0)
```

```
    message = error.get("message", "Unknown error")
```

```
    retryable = code in [-32000, -32603]
```

```
    raise RPCError(f"RPC error {code}: {message}",
        retryable=retryable)
```

```
return data.get("result")
```

```
async def batch_call(self, calls: List[Dict[str, Any]]) -> List[Any]:
    provider = self.manager.select_provider()
```

```
if not provider:
    raise RPCConnectionError("No providers available")
```

```
batch_payload = []
for call in calls:
    self.request_id += 1
    batch_payload.append({
        "jsonrpc": "2.0",
        "id": self.request_id,
        "method": call["method"],
        "params": call.get("params", [])
    })
```

```
try:
    response = await self.client.post(
        provider.url,
        json=batch_payload,
        timeout=60.0
    )
    response.raise_for_status()
```

```
except Exception as e:
    self.manager.record_failure(provider)
    raise
```

```
self.manager.record_success(provider)
```

```
results = response.json()
results.sort(key=lambda r: r["id"])
```

```
return [r.get("result") for r in results]
```

```
async def close(self):
    await self.client.aclose()
```

```
class RPCError(Exception):
    def __init__(self, message: str, retryable: bool = False):
        super().__init__(message)
        self.retryable = retryable
```

```
class RPCConnectionError(Exception):  
    pass
```

The resilient client retries failed requests with exponential backoff, switches providers on failure, and supports batch requests for efficiency.

SECTION 4: RUNNING YOUR OWN NODES

Self-hosted nodes give you complete control, unlimited requests, and lower latency. The tradeoff is operational complexity and infrastructure cost.

Here is a Docker Compose configuration for running an Ethereum execution client with Geth and a consensus client with Lighthouse.

```
version: '3.8'
```

```
services:
```

```
  geth:
```

```
    image: ethereum/client-go:stable
```

```
    container_name: geth
```

```
    restart: unless-stopped
```

```
  ports:
```

```
    - "8545:8545"
```

```
    - "8546:8546"
```

```
    - "30303:30303"
```

```
    - "30303:30303/udp"
```

```
  volumes:
```

```
    - geth-data:/root/.ethereum
```

```
    - ./jwt-secret:/jwt-secret:ro
```

```
  command:
```

```
    - --mainnet
```

```
    - --http
```

```
    - --http.addr = 0.0.0.0
```

```
    - --http.port = 8545
```

```
    - --http.api = eth,net,web3,txpool
```

```
    - --http.vhosts = *
```

- --http.corsdomain = *
- --ws
- --ws.addr = 0.0.0.0
- --ws.port = 8546
- --ws.api = eth,net,web3
- --ws.origins = *
- --authrpc.addr = 0.0.0.0
- --authrpc.port = 8551
- --authrpc.vhosts = *
- --authrpc.jwtsecret = /jwt-secret
- --syncmode = snap
- --cache = 4096
- --maxpeers = 50

healthcheck:

test: ["CMD", "geth", "attach", "--exec", "eth.syncing", "http://localhost:8545"]

interval: 30s

timeout: 10s

retries: 3

logging:

driver: json-file

options:

max-size: "100m"

max-file: "3"

lighthouse:

image: sigp/lighthouse:latest

container_name: lighthouse

restart: unless-stopped

depends_on:

- geth

ports:

- "9000:9000"

- "9000:9000/udp"

- "5052:5052"

volumes:

- lighthouse-data:/root/.lighthouse

- ./jwt-secret:/jwt-secret:ro

command:

- lighthouse

- beacon
- --network = mainnet
- --execution-endpoint = http://geth:8551
- --execution-jwt = /jwt-secret
- --http
- --http-address = 0.0.0.0
- --http-port = 5052
- --checkpoint-sync-url = https://beaconstate.info
- --disable-deposit-contract-sync

healthcheck:

test: ["CMD", "curl", "-f", "http://localhost:5052/eth/v1/node/health"]

interval: 30s

timeout: 10s

retries: 3

logging:

driver: json-file

options:

max-size: "100m"

max-file: "3"

volumes:

geth-data:

lighthouse-data:

Breaking down the critical flags:

--syncmode = snap uses snap sync which downloads state snapshots instead of processing every historical block. This reduces sync time from weeks to hours.

--cache = 4096 allocates 4GB of RAM for the database cache. More cache means faster queries but higher memory usage.

--http.api = eth,net,web3,txpool exposes these RPC namespaces. The txpool namespace is useful for monitoring pending transactions but increases attack surface.

--authrpc.jwtsecret configures JWT authentication between the execution and consensus clients. This is required since the merge.

--checkpoint-sync-url enables checkpoint sync for the beacon chain. The beacon node downloads a recent finalized state instead of syncing from genesis.

SECTION 5: NODE SYNCHRONIZATION MONITORING

A node is useless until it syncs with the network. Monitoring sync status is critical.

```
import asyncio
from datetime import datetime
from typing import Dict, Any, Optional

class NodeSyncMonitor:
    def __init__(self, rpc_client: ResilientRPCClient):
        self.rpc = rpc_client
        self.sync_started_at: Optional[datetime] = None
        self.last_block_time: Optional[datetime] = None

    async def get_sync_status(self) -> Dict[str, Any]:
        syncing = await self.rpc.call("eth_syncing")

        if syncing is False:
            latest_block = await self.rpc.call("eth_getBlockByNumber",
            ["latest", False])
            block_timestamp = int(latest_block["timestamp"], 16)
            block_age = datetime.now().timestamp() - block_timestamp

            return {
                "synced": True,
                "current_block": int(latest_block["number"], 16),
                "block_age_seconds": block_age,
                "healthy": block_age < 60
            }

        current_block = int(syncing["currentBlock"], 16)
        highest_block = int(syncing["highestBlock"], 16)
        starting_block = int(syncing["startingBlock"], 16)
```

```
total_blocks = highest_block - starting_block
synced_blocks = current_block - starting_block
progress = (synced_blocks / total_blocks) * 100 if total_blocks > 0
else 0
```

```
blocks_remaining = highest_block - current_block
```

```
if self.last_block_time:
    time_delta = (datetime.now() - self.last_block_time).total_seconds()
    if time_delta > 0:
        blocks_per_second = 1 / time_delta
        eta_seconds = blocks_remaining / blocks_per_second
    else:
        eta_seconds = None
    else:
        eta_seconds = None
```

```
self.last_block_time = datetime.now()
```

```
return {
    "synced": False,
    "current_block": current_block,
    "highest_block": highest_block,
    "starting_block": starting_block,
    "progress_percent": round(progress, 2),
    "blocks_remaining": blocks_remaining,
    "eta_seconds": eta_seconds
}
```

```
async def wait_for_sync(self, check_interval: int = 30):
    print("Waiting for node to sync...")
    self.sync_started_at = datetime.now()
```

```
while True:
    status = await self.get_sync_status()
```

```
if status["synced"]:
    if status["healthy"]:
        elapsed = (datetime.now() - self.sync_started_at).total_seconds()
```

```

print(f'Node synced in {elapsed:.0f} seconds')
return status
else:
    print(f'Node synced but block is {status["block_age_seconds"]:.0f}s
old")
else:
    progress = status["progress_percent"]
    remaining = status["blocks_remaining"]
    eta = status.get("eta_seconds")

    if eta:
        eta_str = f'ETA: {eta/3600:.1f}h' if eta > 3600 else f'ETA:
{eta/60:.0f}m'
    else:
        eta_str = "ETA: calculating..."

    print(f'Sync progress: {progress:.2f}% ({remaining} blocks
remaining) {eta_str}')

    await asyncio.sleep(check_interval)

    async def get_peer_count(self) -> int:
        result = await self.rpc.call("net_peerCount")
        return int(result, 16)

    async def health_check(self) -> Dict[str, Any]:
        try:
            sync_status = await self.get_sync_status()
            peer_count = await self.get_peer_count()

            chain_id = await self.rpc.call("eth_chainId")
            client_version = await self.rpc.call("web3_clientVersion")

            healthy = (
                sync_status.get("synced", False) and
                sync_status.get("healthy", False) and
                peer_count >= 3
            )

        return {

```

```

"healthy": healthy,
"sync_status": sync_status,
"peer_count": peer_count,
"chain_id": int(chain_id, 16),
"client_version": client_version,
"timestamp": datetime.now().isoformat()
}

```

```

except Exception as e:
    return {
        "healthy": False,
        "error": str(e),
        "timestamp": datetime.now().isoformat()
    }

```

The sync monitor tracks synchronization progress, estimates completion time, and performs health checks. A node is only healthy when synced and receiving new blocks.

SECTION 6: LOAD BALANCING MULTIPLE NODES

For high availability, run multiple nodes behind a load balancer. Here is an Nginx configuration.

```

upstream ethereum_nodes {
    least_conn;

    server node1.internal:8545 weight=5 max_fails=3
    fail_timeout=30s;
    server node2.internal:8545 weight=5 max_fails=3
    fail_timeout=30s;
    server node3.internal:8545 weight=5 max_fails=3
    fail_timeout=30s backup;
}

server {
    listen 8545;

    location / {

```

```
proxy_pass http://ethereum_nodes;
proxy_http_version 1.1;

proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

proxy_connect_timeout 10s;
proxy_send_timeout 60s;
proxy_read_timeout 60s;

proxy_next_upstream error timeout http_500 http_502 http_503
http_504;
proxy_next_upstream_tries 3;
proxy_next_upstream_timeout 30s;

proxy_buffer_size 128k;
proxy_buffers 4 256k;
proxy_busy_buffers_size 256k;

limit_req zone=rpc_limit burst=50 nodelay;
}

location /health {
    access_log off;
    return 200 "healthy\n";
    add_header Content-Type text/plain;
}
}
```

```
limit_req_zone $binary_remote_addr zone=rpc_limit:10m
rate=100r/s;
```

Breaking down the configuration:

least_conn directs traffic to the node with fewest active connections.
This distributes load evenly.

max_fails=3 fail_timeout=30s marks a node as unhealthy after 3 failures and excludes it for 30 seconds.

proxy_next_upstream automatically retries on the next node when the current one fails.

limit_req_zone protects against request flooding with rate limiting.

SECTION 7: WEBSOCKET CONNECTIONS FOR REAL-TIME DATA

WebSocket connections provide real-time event streaming. Managing these connections at scale requires careful handling.

```
import asyncio
import json
from typing import Callable, Dict, Any, Set
from dataclasses import dataclass, field
import websockets

@dataclass
class Subscription:
    id: str
    event_type: str
    callback: Callable
    params: Dict[str, Any] = field(default_factory=dict)

class WebSocketRPCClient:
    def __init__(self, ws_urls: list[str]):
        self.ws_urls = ws_urls
        self.current_url_index = 0
        self.websocket = None
        self.subscriptions: Dict[str, Subscription] = {}
        self.pending_requests: Dict[int, asyncio.Future] = {}
        self.request_id = 0
        self.connected = False
        self.reconnect_delay = 1.0
        self.max_reconnect_delay = 60.0
        self.ping_interval = 30
        self.receive_task = None
        self.ping_task = None
```

```
async def connect(self):
    while True:
        url = self.ws_urls[self.current_url_index]

        try:
            self.websocket = await websockets.connect(
                url,
                ping_interval=self.ping_interval,
                ping_timeout=10,
                close_timeout=5
            )

            self.connected = True
            self.reconnect_delay = 1.0

            self.receive_task = asyncio.create_task(self._receive_loop())

            await self._resubscribe_all()

            return

        except Exception as e:
            print(f"Connection to {url} failed: {e}")

            self.current_url_index = (self.current_url_index + 1) %
            len(self.ws_urls)

            await asyncio.sleep(self.reconnect_delay)
            self.reconnect_delay = min(
                self.reconnect_delay * 2,
                self.max_reconnect_delay
            )

    async def _receive_loop(self):
        try:
            async for message in self.websocket:
                data = json.loads(message)
                await self._handle_message(data)

        except websockets.ConnectionClosed:
```

```
self.connected = False
await self._handle_disconnect()
```

```
except Exception as e:
    print(f'Receive error: {e}')
    self.connected = False
    await self._handle_disconnect()
```

```
async def _handle_message(self, data: Dict[str, Any]):
    if "id" in data and data["id"] in self.pending_requests:
        future = self.pending_requests.pop(data["id"])
```

```
        if "error" in data:
            future.set_exception(RPCError(data["error"]["message"]))
        else:
            future.set_result(data.get("result"))
        return
```

```
        if "method" in data and data["method"] == "eth_subscription":
            params = data.get("params", {})
            sub_id = params.get("subscription")
```

```
            if sub_id in self.subscriptions:
                sub = self.subscriptions[sub_id]
                result = params.get("result")
```

```
            try:
                if asyncio.iscoroutinefunction(sub.callback):
                    await sub.callback(result)
            else:
                sub.callback(result)
            except Exception as e:
                print(f'Subscription callback error: {e}')
```

```
        async def _handle_disconnect(self):
            for future in self.pending_requests.values():
                future.set_exception(RPCConnectionError("Connection lost"))
            self.pending_requests.clear()
```

```
        await self.connect()
```



```

async def _resubscribe_all(self):
    old_subscriptions = list(self.subscriptions.values())
    self.subscriptions.clear()

    for sub in old_subscriptions:
        await self.subscribe(sub.event_type, sub.callback, sub.params)

    async def call(self, method: str, params: list = None) -> Any:
        if not self.connected:
            raise RPCConnectionError("Not connected")

        self.request_id += 1
        request_id = self.request_id

        payload = {
            "jsonrpc": "2.0",
            "id": request_id,
            "method": method,
            "params": params or []
        }

        future = asyncio.get_event_loop().create_future()
        self.pending_requests[request_id] = future

        try:
            await self.websocket.send(json.dumps(payload))
            result = await asyncio.wait_for(future, timeout=30.0)
            return result

        except asyncio.TimeoutError:
            self.pending_requests.pop(request_id, None)
            raise RPCError("Request timeout", retryable=True)

    async def subscribe(
        self,
        event_type: str,
        callback: Callable,
        params: Dict[str, Any] = None
    ) -> str:

```

```
sub_params = [event_type]
if params:
    sub_params.append(params)
```

```
sub_id = await self.call("eth_subscribe", sub_params)
```

```
self.subscriptions[sub_id] = Subscription(
    id=sub_id,
    event_type=event_type,
    callback=callback,
    params=params or {}
)
```

```
return sub_id
```

```
async def unsubscribe(self, subscription_id: str) -> bool:
    result = await self.call("eth_unsubscribe", [subscription_id])
```

```
if result:
    self.subscriptions.pop(subscription_id, None)
```

```
return result
```

```
async def subscribe_new_blocks(self, callback: Callable) -> str:
    return await self.subscribe("newHeads", callback)
```

```
async def subscribe_pending_transactions(self, callback: Callable) -
> str:
    return await self.subscribe("newPendingTransactions", callback)
```

```
async def subscribe_logs(
    self,
    callback: Callable,
    address: str = None,
    topics: list = None
) -> str:
```

```
params = {}
if address:
```

```

params["address"] = address
if topics:
    params["topics"] = topics

return await self.subscribe("logs", callback, params)

async def close(self):
    self.connected = False

    if self.receive_task:
        self.receive_task.cancel()

    if self.websocket:
        await self.websocket.close()

```

The WebSocket client manages subscriptions, handles reconnection, and automatically resubscribes after connection loss. This is essential for applications that need real-time blockchain events.

SECTION 8: ARCHIVE NODE ACCESS PATTERNS

Archive nodes are expensive but necessary for historical queries. Optimize their usage with smart access patterns.

```

from datetime import datetime, timedelta
from typing import Optional, Tuple
import bisect

class ArchiveNodeRouter:
    def __init__(
        self,
        full_node_client: ResilientRPCClient,
        archive_node_client: ResilientRPCClient,
        archive_block_threshold: int = 128
    ):
        self.full_node = full_node_client
        self.archive_node = archive_node_client
        self.archive_threshold = archive_block_threshold
        self.current_block = 0

```

```
self.last_block_update = 0
```

```
async def update_current_block(self):  
    now = datetime.now().timestamp()
```

```
    if now - self.last_block_update < 12:  
        return
```

```
    result = await self.full_node.call("eth_blockNumber")  
    self.current_block = int(result, 16)  
    self.last_block_update = now
```

```
    def needs_archive(self, block_param: str) -> bool:  
        if block_param in ["latest", "pending", "safe", "finalized"]:  
            return False
```

```
        if block_param == "earliest":  
            return True
```

```
        if block_param.startswith("0x"):  
            block_number = int(block_param, 16)  
        else:  
            block_number = int(block_param)
```

```
        blocks_behind = self.current_block - block_number  
        return blocks_behind > self.archive_threshold
```

```
    async def call(self, method: str, params: list = None) -> Any:  
        await self.update_current_block()
```

```
        block_param = self._extract_block_param(method, params or [])
```

```
        if block_param and self.needs_archive(block_param):  
            return await self.archive_node.call(method, params)  
        else:  
            return await self.full_node.call(method, params)
```

```
    def _extract_block_param(self, method: str, params: list) ->  
Optional[str]:  
        block_param_methods = {
```

```

"eth_getBalance": 1,
"eth_getCode": 1,
"eth_getTransactionCount": 1,
"eth_getStorageAt": 2,
"eth_call": 1,
"eth_getBlockByNumber": 0,
"eth_getBlockTransactionCountByNumber": 0,
"eth_getUncleCountByBlockNumber": 0,
"eth_getTransactionByBlockNumberAndIndex": 0,
"eth_getUncleByBlockNumberAndIndex": 0,
}

```

```

if method not in block_param_methods:
    return None

```

```

param_index = block_param_methods[method]

```

```

if len(params) > param_index:
    return params[param_index]

```

```

return None

```

```

class HistoricalDataFetcher:
    def __init__(self, archive_client: ResilientRPCClient):
        self.client = archive_client
        self.block_timestamp_cache: Dict[int, int] = {}

```

```

    async def get_block_by_timestamp(
        self,
        target_timestamp: int,
        before: bool = True
    ) -> int:

```

```

        latest = await self.client.call("eth_blockNumber")
        latest_num = int(latest, 16)

```

```

        latest_block = await self.client.call(
            "eth_getBlockByNumber",
            [hex(latest_num), False]
        )

```

```

latest_ts = int(latest_block["timestamp"], 16)

genesis_block = await self.client.call(
    "eth_getBlockByNumber",
    ["0x0", False]
)
genesis_ts = int(genesis_block["timestamp"], 16)

if target_timestamp >= latest_ts:
    return latest_num
if target_timestamp <= genesis_ts:
    return 0

low = 0
high = latest_num

while low < high:
    mid = (low + high) // 2

    if mid in self.block_timestamp_cache:
        mid_ts = self.block_timestamp_cache[mid]
    else:
        mid_block = await self.client.call(
            "eth_getBlockByNumber",
            [hex(mid), False]
        )
        mid_ts = int(mid_block["timestamp"], 16)
    self.block_timestamp_cache[mid] = mid_ts

    if mid_ts < target_timestamp:
        low = mid + 1
    else:
        high = mid

    if before:
        while low > 0:
            if low in self.block_timestamp_cache:
                block_ts = self.block_timestamp_cache[low]
            else:
                block = await self.client.call(

```

```
"eth_getBlockByNumber",  
[hex(low), False]  
)  
block_ts = int(block["timestamp"], 16)  
self.block_timestamp_cache[low] = block_ts
```

```
if block_ts <= target_timestamp:  
    return low  
    low -= 1
```

```
return low
```

```
async def get_historical_balance(  
    self,  
    address: str,  
    timestamp: int  
) -> Tuple[int, int]:
```

```
    block = await self.get_block_by_timestamp(timestamp)
```

```
    balance = await self.client.call(  
        "eth_getBalance",  
        [address, hex(block)]  
    )
```

```
    return int(balance, 16), block
```

```
async def trace_balance_history(  
    self,  
    address: str,  
    start_timestamp: int,  
    end_timestamp: int,  
    interval_seconds: int = 86400  
) -> list:
```

```
    history = []  
    current_ts = start_timestamp
```

```
    while current_ts <= end_timestamp:  
        balance, block = await self.get_historical_balance(address,
```

```
current_ts)
```

```
history.append({  
    "timestamp": current_ts,  
    "block": block,  
    "balance": balance  
})
```

```
current_ts += interval_seconds
```

```
return history
```

The archive router automatically directs queries to the appropriate node type. Historical data fetcher provides utilities for timestamp-based queries that archive nodes enable.

SECTION 9: MULTICHAIN RPC MANAGEMENT

Production applications often interact with multiple chains. Managing RPC connections across chains requires organization.

```
from enum import Enum  
from typing import Dict, Optional
```

```
class Chain(Enum):  
    ETHEREUM = 1  
    POLYGON = 137  
    ARBITRUM = 42161  
    OPTIMISM = 10  
    BASE = 8453  
    BSC = 56  
    AVALANCHE = 43114
```

```
CHAIN_CONFIGS = {  
    Chain.ETHEREUM: {  
        "name": "Ethereum",  
        "native_currency": "ETH",  
        "block_time": 12,  
        "providers": [  

```



```

{"name": "alchemy", "url": "https://eth-mainnet.g.alchemy.com/v2/
{api_key}"},
{"name": "infura", "url": "https://mainnet.infura.io/v3/{api_key}"},
{"name": "quicknode", "url": "https://{subdomain}.quiknode.pro/
{api_key}"},
],
},
Chain.POLYGON: {
"name": "Polygon",
"native_currency": "MATIC",
"block_time": 2,
"providers": [
{"name": "alchemy", "url": "https://polygon-
mainnet.g.alchemy.com/v2/{api_key}"},
{"name": "infura", "url": "https://polygon-mainnet.infura.io/v3/
{api_key}"},
],
},
Chain.ARBITRUM: {
"name": "Arbitrum One",
"native_currency": "ETH",
"block_time": 0.25,
"providers": [
{"name": "alchemy", "url": "https://arb-mainnet.g.alchemy.com/v2/
{api_key}"},
{"name": "infura", "url": "https://arbitrum-mainnet.infura.io/v3/
{api_key}"},
],
},
Chain.OPTIMISM: {
"name": "Optimism",
"native_currency": "ETH",
"block_time": 2,
"providers": [
{"name": "alchemy", "url": "https://opt-mainnet.g.alchemy.com/v2/
{api_key}"},
{"name": "infura", "url": "https://optimism-mainnet.infura.io/v3/
{api_key}"},
],
},

```

```
Chain.BASE: {
    "name": "Base",
    "native_currency": "ETH",
    "block_time": 2,
    "providers": [
        {"name": "alchemy", "url": "https://base-mainnet.g.alchemy.com/
v2/{api_key}"}
    ]
}
}
```

```
class MultiChainRPCManager:
    def __init__(self, api_keys: Dict[str, str]):
        self.api_keys = api_keys
        self.clients: Dict[Chain, ResilientRPCClient] = {}

    def _build_provider_url(self, url_template: str) -> str:
        url = url_template
        for key, value in self.api_keys.items():
            url = url.replace("{} + key + {}", value)
        return url

    def get_client(self, chain: Chain) -> ResilientRPCClient:
        if chain in self.clients:
            return self.clients[chain]

        config = CHAIN_CONFIGS.get(chain)
        if not config:
            raise ValueError(f'Unknown chain: {chain}')

        provider_configs = []
        for i, provider in enumerate(config["providers"]):
            url = self._build_provider_url(provider["url"])

            if "{" in url:
                continue

            provider_configs.append({
                "name": f'{chain.name}_{provider["name"]}',
                "url": url,
```

```
"priority": i,  
"weight": 1,  
"rps_limit": 25  
})
```

```
if not provider_configs:  
    raise ValueError(f'No valid providers for {chain}')
```

```
client = ResilientRPCClient(provider_configs)  
self.clients[chain] = client
```

```
return client
```

```
async def call(  
    self,  
    chain: Chain,  
    method: str,  
    params: list = None  
    ) -> Any:  
    client = self.get_client(chain)  
    return await client.call(method, params)
```

```
async def multi_chain_call(  
    self,  
    chains: list[Chain],  
    method: str,  
    params: list = None  
    ) -> Dict[Chain, Any]:
```

```
    tasks = {}  
    for chain in chains:  
        tasks[chain] = self.call(chain, method, params)
```

```
    results = {}  
    for chain, task in tasks.items():  
        try:  
            results[chain] = await task  
        except Exception as e:  
            results[chain] = {"error": str(e)}
```

```
return results
```

```
async def get_gas_prices_all_chains(self) -> Dict[Chain, str]:  
    chains = list(CHAIN_CONFIGS.keys())  
    results = await self.multi_chain_call(chains, "eth_gasPrice")
```

```
    formatted = {}  
    for chain, result in results.items():  
        if isinstance(result, str):  
            gwei = int(result, 16) / 1e9  
            formatted[chain] = f"{gwei:.2f} gwei"  
        else:  
            formatted[chain] = "error"
```

```
    return formatted
```

```
async def close_all(self):  
    for client in self.clients.values():  
        await client.close()
```

The multichain manager provides unified access to multiple blockchain networks. Cross-chain applications can query all chains with a single interface.

SECTION 10: PRODUCTION DEPLOYMENT CHECKLIST

Before deploying RPC infrastructure to production, verify these requirements.

This is the deployment verification script.

```
import asyncio  
from typing import Dict, Any, List  
  
class RPCDeploymentVerifier:  
    def __init__(self, client: ResilientRPCClient):  
        self.client = client  
        self.checks: List[Dict[str, Any]] = []
```

```
async def run_all_checks(self) -> Dict[str, Any]:
    self.checks = []
```

```
    await self._check_connectivity()
    await self._check_sync_status()
    await self._check_peer_count()
    await self._check_chain_id()
    await self._check_latency()
    await self._check_rate_limits()
    await self._check_required_methods()
```

```
    passed = sum(1 for c in self.checks if c["passed"])
    total = len(self.checks)
```

```
    return {
        "passed": passed == total,
        "summary": f"{passed}/{total} checks passed",
        "checks": self.checks
    }
```

```
    async def _check_connectivity(self):
        check = {"name": "connectivity", "passed": False}
```

```
        try:
            result = await self.client.call("web3_clientVersion")
            check["passed"] = True
            check["client_version"] = result
        except Exception as e:
            check["error"] = str(e)
```

```
        self.checks.append(check)
```

```
    async def _check_sync_status(self):
        check = {"name": "sync_status", "passed": False}
```

```
        try:
            syncing = await self.client.call("eth_syncing")
```

```
            if syncing is False:
                latest = await self.client.call("eth_getBlockByNumber", ["latest",
```

```

False])
block_ts = int(latest["timestamp"], 16)
age = datetime.now().timestamp() - block_ts

check["passed"] = age < 60
check["block_age_seconds"] = age
check["synced"] = True
else:
current = int(syncing["currentBlock"], 16)
highest = int(syncing["highestBlock"], 16)
check["progress"] = f"{{(current/highest)*100:.2f}}%"
check["synced"] = False

except Exception as e:
check["error"] = str(e)

self.checks.append(check)

async def _check_peer_count(self):
check = {"name": "peer_count", "passed": False}

try:
result = await self.client.call("net_peerCount")
peers = int(result, 16)

check["peer_count"] = peers
check["passed"] = peers >= 3

except Exception as e:
check["error"] = str(e)

self.checks.append(check)

async def _check_chain_id(self):
check = {"name": "chain_id", "passed": False}

try:
result = await self.client.call("eth_chainId")
chain_id = int(result, 16)

```

```
check["chain_id"] = chain_id
check["passed"] = True
```

```
except Exception as e:
    check["error"] = str(e)
```

```
self.checks.append(check)
```

```
async def _check_latency(self):
    check = {"name": "latency", "passed": False}
```

```
try:
    latencies = []
```

```
for _ in range(5):
    start = datetime.now()
    await self.client.call("eth_blockNumber")
    elapsed = (datetime.now() - start).total_seconds() * 1000
    latencies.append(elapsed)
```

```
avg_latency = sum(latencies) / len(latencies)
max_latency = max(latencies)
```

```
check["avg_latency_ms"] = round(avg_latency, 2)
check["max_latency_ms"] = round(max_latency, 2)
check["passed"] = avg_latency < 500 and max_latency < 2000
```

```
except Exception as e:
    check["error"] = str(e)
```

```
self.checks.append(check)
```

```
async def _check_rate_limits(self):
    check = {"name": "rate_limits", "passed": False}
```

```
try:
    successful = 0
    rate_limited = 0
```

```
tasks = [
```

```
self.client.call("eth_blockNumber")
for _ in range(50)
]
```

```
results = await asyncio.gather(*tasks, return_exceptions=True)
```

```
for result in results:
    if isinstance(result, Exception):
        if "rate" in str(result).lower():
            rate_limited += 1
        else:
            successful += 1
```

```
check["successful"] = successful
check["rate_limited"] = rate_limited
check["passed"] = rate_limited == 0
```

```
except Exception as e:
    check["error"] = str(e)
```

```
self.checks.append(check)
```

```
async def _check_required_methods(self):
```

```
    required_methods = [
```

```
        ("eth_blockNumber", []),
```

```
        ("eth_getBalance",
```

```
        ["0x0000000000000000000000000000000000000000000000000000000000000000", "latest"]),
```

```
        ("eth_getBlockByNumber", ["latest", False]),
```

```
        ("eth_call", [{"to":
```

```
        "0x0000000000000000000000000000000000000000000000000000000000000000", "data":
```

```
        "0x"}], "latest")),
```

```
        ("eth_estimateGas", [{"to":
```

```
        "0x0000000000000000000000000000000000000000000000000000000000000000"}])),
```

```
        ("eth_sendRawTransaction", None),
```

```
        ("eth_getTransactionReceipt",
```

```
        ["0x0000000000000000000000000000000000000000000000000000000000000000",
```

```
        ("eth_getLogs", [{"fromBlock": "latest", "toBlock": "latest"}])),
```

```
    ]
```

```
for method, params in required_methods:
```



```
check = {"name": f"method_{method}", "passed": False}
```

```
try:
```

```
if params is None:
```

```
check["passed"] = True
```

```
check["note"] = "Skipped (would modify state)"
```

```
else:
```

```
await self.client.call(method, params)
```

```
check["passed"] = True
```

```
except RPCError as e:
```

```
if "not found" in str(e).lower():
```

```
check["error"] = "Method not supported"
```

```
else:
```

```
check["passed"] = True
```

```
except Exception as e:
```

```
check["error"] = str(e)
```

```
self.checks.append(check)
```

```
async def verify_production_deployment():
```

```
providers = [
```

```
{ "name": "primary", "url": os.environ["RPC_URL_PRIMARY"],
```

```
"priority": 0},
```

```
{ "name": "secondary", "url": os.environ["RPC_URL_SECONDARY"],
```

```
"priority": 1}
```

```
]
```

```
client = ResilientRPCClient(providers)
```

```
verifier = RPCDeploymentVerifier(client)
```

```
results = await verifier.run_all_checks()
```

```
print(f"Deployment verification: {results['summary']}")
```

```
for check in results["checks"]:
```

```
status = "PASS" if check["passed"] else "FAIL"
```

```
print(f" [{status}] {check['name']}")
```

```
if "error" in check:
```

```
print(f" Error: {check['error']}")
```

```
await client.close()
```

```
return results["passed"]
```

The deployment verifier confirms connectivity, synchronization, peer count, latency, rate limits, and method availability before traffic hits your infrastructure.

This chapter provided the foundation for production RPC infrastructure. You learned provider management with automatic failover, building resilient clients with retry logic, running your own nodes with Docker, monitoring synchronization and health, load balancing across multiple nodes, WebSocket management for real-time data, archive node access patterns, and multichain organization. The next chapter builds on this foundation with event indexing and data pipelines.

CHAPTER 2: EVENT INDEXING AND DATA PIPELINES

Blockchain data is not directly queryable. You cannot ask the blockchain for all transfers to a specific address or the total volume traded on a DEX. The blockchain stores raw transaction data and event logs. To answer useful questions, you must index this data into a queryable database. This chapter teaches you how.

SECTION 1: WHY INDEXING MATTERS

The blockchain is an append-only log of transactions. Each block contains transactions. Each transaction may emit events (logs). To find specific information, you must scan through all blocks and filter for what you need.

Consider this problem: find all ERC20 transfers to address X. Without an index, you must download every block since the token contract deployed, decode every transaction and log, filter for Transfer events with the recipient matching X, and repeat this for every query.

With an index, you query your database: `SELECT FROM transfers`

WHERE to_address = X. The response comes in milliseconds instead of hours.

Indexing transforms blockchain data from a linear log into a structured database. This is essential for any application that needs to query historical data, display user activity, calculate analytics, or aggregate information across transactions.

SECTION 2: INDEXER ARCHITECTURE

A production indexer has these components:

The block fetcher retrieves blocks from the RPC node. It must handle rate limits, retries, and batch requests efficiently.

The event decoder parses raw log data into structured events. It needs contract ABIs to decode event parameters.

The data transformer converts decoded events into your application's data model. A Transfer event becomes a row in your transfers table.

The database writer persists transformed data. It must handle duplicates, ordering, and atomic writes.

The chain reorganization handler detects and reverses changes when the chain reorganizes. This is critical for data integrity.

The checkpoint manager tracks indexing progress and enables resume after restarts.

Here is the core indexer structure.

```
import asyncio
from dataclasses import dataclass, field
from typing import Dict, List, Any, Optional, Callable
from datetime import datetime
import json
```

```
@dataclass
class IndexerConfig:
    start_block: int
    contracts: Dict[str, str]
    batch_size: int = 100
    confirmations: int = 12
    checkpoint_interval: int = 1000
```

```
@dataclass
class DecodedEvent:
    block_number: int
    block_timestamp: int
    transaction_hash: str
    log_index: int
    contract_address: str
    event_name: str
    args: Dict[str, Any]
    raw_data: str
    raw_topics: List[str]
```

```
@dataclass
class IndexerState:
    last_indexed_block: int
    last_confirmed_block: int
    pending_blocks: Dict[int, List[DecodedEvent]] =
field(default_factory = dict)
    reorg_depth: int = 0
```

```
class BlockchainIndexer:
    def __init__(
        self,
        rpc_client,
        database,
        config: IndexerConfig
    ):
        self.rpc = rpc_client
        self.db = database
        self.config = config
        self.state = IndexerState(
            last_indexed_block = config.start_block - 1,
```

```

last_confirmed_block = config.start_block - 1
)
self.event_decoders: Dict[str, Any] = {}
self.event_handlers: Dict[str, List[Callable]] = {}
self.running = False

def register_contract(self, address: str, abi: List[Dict]):
    address = address.lower()
    decoder = EventDecoder(abi)
    self.event_decoders[address] = decoder

def register_handler(self, event_name: str, handler: Callable):
    if event_name not in self.event_handlers:
        self.event_handlers[event_name] = []
    self.event_handlers[event_name].append(handler)

async def start(self):
    self.running = True
    await self._load_checkpoint()

    while self.running:
        try:
            await self._index_batch()
            await self._process_confirmations()
            await self._check_reorgs()
            await self._save_checkpoint()

        except Exception as e:
            print(f"Indexer error: {e}")
            await asyncio.sleep(5)

    async def stop(self):
        self.running = False
        await self._save_checkpoint()

    async def _load_checkpoint(self):
        checkpoint = await self.db.get_checkpoint()

        if checkpoint:
            self.state.last_indexed_block = checkpoint["last_indexed_block"]

```

```
self.state.last_confirmed_block =  
checkpoint["last_confirmed_block"]  
print(f'Resumed from block {self.state.last_indexed_block}')  
else:  
print(f'Starting fresh from block {self.config.start_block}')
```

```
async def _save_checkpoint(self):  
    await self.db.save_checkpoint({  
        "last_indexed_block": self.state.last_indexed_block,  
        "last_confirmed_block": self.state.last_confirmed_block,  
        "timestamp": datetime.now().isoformat()  
    })
```

```
async def _index_batch(self):  
    current_block = await self._get_current_block()  
    target_block = min(  
        self.state.last_indexed_block + self.config.batch_size,  
        current_block  
    )
```

```
if target_block <= self.state.last_indexed_block:  
    await asyncio.sleep(1)  
    return
```

```
from_block = self.state.last_indexed_block + 1  
to_block = target_block
```

```
logs = await self._fetch_logs(from_block, to_block)  
blocks = await self._fetch_blocks(from_block, to_block)
```

```
block_timestamps = {  
    int(b["number"], 16): int(b["timestamp"], 16)  
    for b in blocks  
}
```

```
events = self._decode_logs(logs, block_timestamps)
```

```
for block_num in range(from_block, to_block + 1):  
    block_events = [e for e in events if e.block_number ==  
                    block_num]
```

```
self.state.pending_blocks[block_num] = block_events
```

```
self.state.last_indexed_block = to_block
```

```
print(f"Indexed blocks {from_block} to {to_block} ({len(events)}  
events)")
```

```
async def _get_current_block(self) -> int:  
    result = await self.rpc.call("eth_blockNumber")  
    return int(result, 16)
```

```
async def _fetch_logs(self, from_block: int, to_block: int) ->  
List[Dict]:  
    contract_addresses = list(self.event_decoders.keys())
```

```
    logs = await self.rpc.call("eth_getLogs", [{  
        "fromBlock": hex(from_block),  
        "toBlock": hex(to_block),  
        "address": contract_addresses  
    }])
```

```
    return logs
```

```
async def _fetch_blocks(self, from_block: int, to_block: int) ->  
List[Dict]:  
    calls = [  
        {"method": "eth_getBlockByNumber", "params": [hex(b), False]}  
        for b in range(from_block, to_block + 1)  
    ]
```

```
    return await self.rpc.batch_call(calls)
```

```
def _decode_logs(  
    self,  
    logs: List[Dict],  
    block_timestamps: Dict[int, int]  
) -> List[DecodedEvent]:
```

```
    events = []
```

```

for log in logs:
    address = log["address"].lower()

    if address not in self.event_decoders:
        continue

    decoder = self.event_decoders[address]
    decoded = decoder.decode_log(log)

    if decoded:
        block_num = int(log["blockNumber"], 16)

        event = DecodedEvent(
            block_number=block_num,
            block_timestamp=block_timestamps.get(block_num, 0),
            transaction_hash=log["transactionHash"],
            log_index=int(log["logIndex"], 16),
            contract_address=address,
            event_name=decoded["name"],
            args=decoded["args"],
            raw_data=log["data"],
            raw_topics=log["topics"]
        )

        events.append(event)

    return events

    async def _process_confirmations(self):
        current_block = await self._get_current_block()
        confirmed_block = current_block - self.config.confirmations

        blocks_to_confirm = []
        for block_num in sorted(self.state.pending_blocks.keys()):
            if block_num <= confirmed_block:
                blocks_to_confirm.append(block_num)

        for block_num in blocks_to_confirm:
            events = self.state.pending_blocks.pop(block_num)

```



```

for event in events:
    await self._handle_event(event)

self.state.last_confirmed_block = block_num

async def _handle_event(self, event: DecodedEvent):
    handlers = self.event_handlers.get(event.event_name, [])

    for handler in handlers:
        try:
            if asyncio.iscoroutinefunction(handler):
                await handler(event)
            else:
                handler(event)
        except Exception as e:
            print(f"Handler error for {event.event_name}: {e}")

    async def _check_reorgs(self):
        if not self.state.pending_blocks:
            return

        oldest_pending = min(self.state.pending_blocks.keys())

        block = await self.rpc.call("eth_getBlockByNumber",
            [hex(oldest_pending), False])

        if block is None:
            await self._handle_reorg(oldest_pending)

    async def _handle_reorg(self, from_block: int):
        print(f"Chain reorg detected at block {from_block}")

        self.state.reorg_depth += 1

        for block_num in list(self.state.pending_blocks.keys()):
            if block_num >= from_block:
                del self.state.pending_blocks[block_num]

        await self.db.delete_events_from_block(from_block)

```

```
self.state.last_indexed_block = from_block - 1
```

```
if self.state.last_confirmed_block >= from_block:  
self.state.last_confirmed_block = from_block - 1
```

SECTION 3: EVENT DECODING

Raw blockchain logs are encoded. You need the contract ABI to decode them into structured data.

```
from eth_abi import decode  
from eth_utils import event_abi_to_log_topic, keccak  
import json
```

```
class EventDecoder:  
    def __init__(self, abi: List[Dict]):  
        self.events = {}  
        self.topic_to_event = {}
```

```
    for item in abi:  
        if item.get("type") == "event":  
            name = item["name"]  
            inputs = item.get("inputs", [])
```

```
            signature = self._build_signature(name, inputs)  
            topic = "0x" + keccak(text=signature).hex()
```

```
            self.events[name] = {  
                "name": name,  
                "inputs": inputs,  
                "signature": signature,  
                "topic": topic  
            }
```

```
            self.topic_to_event[topic] = self.events[name]
```

```
    def _build_signature(self, name: str, inputs: List[Dict]) -> str:  
        types = []  
        for inp in inputs:
```

```
types.append(self._get_canonical_type(inp["type"]))
return f'{name}({','.join(types)})'
```

```
def _get_canonical_type(self, type_str: str) -> str:
    if type_str == "uint":
        return "uint256"
    if type_str == "int":
        return "int256"
    return type_str
```

```
def decode_log(self, log: Dict) -> Optional[Dict]:
    topics = log.get("topics", [])
```

```
    if not topics:
        return None
```

```
    topic0 = topics[0]
```

```
    if topic0 not in self.topic_to_event:
        return None
```

```
    event_def = self.topic_to_event[topic0]
```

```
    indexed_inputs = []
    non_indexed_inputs = []
```

```
    for inp in event_def["inputs"]:
        if inp.get("indexed"):
            indexed_inputs.append(inp)
        else:
            non_indexed_inputs.append(inp)
```

```
    args = {}
```

```
    for i, inp in enumerate(indexed_inputs):
        if i + 1 < len(topics):
            topic_data = topics[i + 1]
            value = self._decode_indexed(inp["type"], topic_data)
            args[inp["name"]] = value
```

```

if non_indexed_inputs and log.get("data") and log["data"] != "0x":
    data_types = [inp["type"] for inp in non_indexed_inputs]

    try:
        data_bytes = bytes.fromhex(log["data"][2:])
        decoded_data = decode(data_types, data_bytes)

        for i, inp in enumerate(non_indexed_inputs):
            args[inp["name"]] = self._convert_value(decoded_data[i])

    except Exception as e:
        print(f"Failed to decode log data: {e}")

    return {
        "name": event_def["name"],
        "args": args
    }

def _decode_indexed(self, type_str: str, topic: str) -> Any:
    topic_bytes = bytes.fromhex(topic[2:])

    if type_str in ["address"]:
        return "0x" + topic_bytes[-20:].hex()

    if type_str.startswith("uint") or type_str.startswith("int"):
        return int.from_bytes(topic_bytes, "big")

    if type_str == "bool":
        return topic_bytes[-1] != 0

    if type_str.startswith("bytes"):
        return "0x" + topic_bytes.hex()

    return "0x" + topic_bytes.hex()

def _convert_value(self, value: Any) -> Any:
    if isinstance(value, bytes):
        return "0x" + value.hex()
    if isinstance(value, tuple):
        return [self._convert_value(v) for v in value]

```

return value

```
ERC20_ABI = [  
  {  
    "type": "event",  
    "name": "Transfer",  
    "inputs": [  
      {"name": "from", "type": "address", "indexed": True},  
      {"name": "to", "type": "address", "indexed": True},  
      {"name": "value", "type": "uint256", "indexed": False}  
    ]  
  },  
  {  
    "type": "event",  
    "name": "Approval",  
    "inputs": [  
      {"name": "owner", "type": "address", "indexed": True},  
      {"name": "spender", "type": "address", "indexed": True},  
      {"name": "value", "type": "uint256", "indexed": False}  
    ]  
  }  
]  
  
ERC721_ABI = [  
  {  
    "type": "event",  
    "name": "Transfer",  
    "inputs": [  
      {"name": "from", "type": "address", "indexed": True},  
      {"name": "to", "type": "address", "indexed": True},  
      {"name": "tokenId", "type": "uint256", "indexed": True}  
    ]  
  },  
  {  
    "type": "event",  
    "name": "Approval",  
    "inputs": [  
      {"name": "owner", "type": "address", "indexed": True},  
      {"name": "approved", "type": "address", "indexed": True},
```

```

{"name": "tokenId", "type": "uint256", "indexed": True}
]
},
{
  "type": "event",
  "name": "ApprovalForAll",
  "inputs": [
    {"name": "owner", "type": "address", "indexed": True},
    {"name": "operator", "type": "address", "indexed": True},
    {"name": "approved", "type": "bool", "indexed": False}
  ]
}
]

```

The event decoder handles ABI parsing, topic matching, and parameter decoding. It converts raw hex data into readable values with named parameters.

SECTION 4: DATABASE SCHEMA DESIGN

Blockchain data has unique characteristics that affect schema design. Blocks are immutable once confirmed. Events are ordered by block, transaction index, and log index. Addresses appear frequently and should be normalized.

Here is a PostgreSQL schema optimized for blockchain data.

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
```

```

CREATE TABLE blocks (
  block_number BIGINT PRIMARY KEY,
  block_hash VARCHAR(66) NOT NULL,
  parent_hash VARCHAR(66) NOT NULL,
  timestamp TIMESTAMP NOT NULL,
  transaction_count INTEGER NOT NULL,
  gas_used BIGINT,
  gas_limit BIGINT,
  base_fee_per_gas BIGINT,
  indexed_at TIMESTAMP DEFAULT NOW()
)

```

);

CREATE INDEX idx_blocks_timestamp ON blocks(timestamp);
CREATE INDEX idx_blocks_hash ON blocks(block_hash);

CREATE TABLE addresses (
id SERIAL PRIMARY KEY,
address VARCHAR(42) UNIQUE NOT NULL,
first_seen_block BIGINT,
is_contract BOOLEAN DEFAULT FALSE,
label VARCHAR(255)
);

CREATE INDEX idx_addresses_address ON addresses(address);

CREATE TABLE transactions (
id SERIAL PRIMARY KEY,
transaction_hash VARCHAR(66) UNIQUE NOT NULL,
block_number BIGINT NOT NULL REFERENCES
blocks(block_number),
transaction_index INTEGER NOT NULL,
from_address_id INTEGER REFERENCES addresses(id),
to_address_id INTEGER REFERENCES addresses(id),
value NUMERIC(78, 0),
gas_price BIGINT,
gas_used BIGINT,
input_data TEXT,
status SMALLINT,

UNIQUE(block_number, transaction_index)
);

CREATE INDEX idx_transactions_block ON
transactions(block_number);
CREATE INDEX idx_transactions_from ON
transactions(from_address_id);
CREATE INDEX idx_transactions_to ON transactions(to_address_id);
CREATE INDEX idx_transactions_hash ON
transactions(transaction_hash);

```

CREATE TABLE events (
  id SERIAL PRIMARY KEY,
  block_number BIGINT NOT NULL REFERENCES
blocks(block_number),
  transaction_hash VARCHAR(66) NOT NULL,
  log_index INTEGER NOT NULL,
  contract_address_id INTEGER REFERENCES addresses(id),
  event_name VARCHAR(100) NOT NULL,
  topic0 VARCHAR(66),
  topic1 VARCHAR(66),
  topic2 VARCHAR(66),
  topic3 VARCHAR(66),
  data TEXT,
  decoded_args JSONB,
  indexed_at TIMESTAMP DEFAULT NOW(),

  UNIQUE(block_number, transaction_hash, log_index)
);

```

```

CREATE INDEX idx_events_block ON events(block_number);
CREATE INDEX idx_events_contract ON events(contract_address_id);
CREATE INDEX idx_events_name ON events(event_name);
CREATE INDEX idx_events_topic0 ON events(topic0);
CREATE INDEX idx_events_decoded ON events USING
GIN(decoded_args);

```

```

CREATE TABLE erc20_transfers (
  id SERIAL PRIMARY KEY,
  block_number BIGINT NOT NULL,
  block_timestamp TIMESTAMP NOT NULL,
  transaction_hash VARCHAR(66) NOT NULL,
  log_index INTEGER NOT NULL,
  token_address_id INTEGER REFERENCES addresses(id),
  from_address_id INTEGER REFERENCES addresses(id),
  to_address_id INTEGER REFERENCES addresses(id),
  value NUMERIC(78, 0) NOT NULL,

  UNIQUE(block_number, transaction_hash, log_index)
);

```



```
CREATE INDEX idx_erc20_token ON
erc20_transfers(token_address_id);
CREATE INDEX idx_erc20_from ON
erc20_transfers(from_address_id);
CREATE INDEX idx_erc20_to ON erc20_transfers(to_address_id);
CREATE INDEX idx_erc20_block ON erc20_transfers(block_number);
CREATE INDEX idx_erc20_timestamp ON
erc20_transfers(block_timestamp);
```

```
CREATE TABLE erc721_transfers (
  id SERIAL PRIMARY KEY,
  block_number BIGINT NOT NULL,
  block_timestamp TIMESTAMP NOT NULL,
  transaction_hash VARCHAR(66) NOT NULL,
  log_index INTEGER NOT NULL,
  token_address_id INTEGER REFERENCES addresses(id),
  from_address_id INTEGER REFERENCES addresses(id),
  to_address_id INTEGER REFERENCES addresses(id),
  token_id NUMERIC(78, 0) NOT NULL,
```

```
  UNIQUE(block_number, transaction_hash, log_index)
);
```

```
CREATE INDEX idx_erc721_token ON
erc721_transfers(token_address_id);
CREATE INDEX idx_erc721_from ON
erc721_transfers(from_address_id);
CREATE INDEX idx_erc721_to ON erc721_transfers(to_address_id);
CREATE INDEX idx_erc721_token_id ON
erc721_transfers(token_address_id, token_id);
```

```
CREATE TABLE nft_ownership (
  id SERIAL PRIMARY KEY,
  token_address_id INTEGER REFERENCES addresses(id),
  token_id NUMERIC(78, 0) NOT NULL,
  owner_address_id INTEGER REFERENCES addresses(id),
  acquired_block BIGINT NOT NULL,
  acquired_timestamp TIMESTAMP NOT NULL,
```

```
  UNIQUE(token_address_id, token_id)
```

```
);
```

```
CREATE INDEX idx_nft_ownership_owner ON  
nft_ownership(owner_address_id);
```

```
CREATE TABLE indexer_checkpoints (  
  id SERIAL PRIMARY KEY,  
  indexer_name VARCHAR(100) UNIQUE NOT NULL,  
  last_indexed_block BIGINT NOT NULL,  
  last_confirmed_block BIGINT NOT NULL,  
  updated_at TIMESTAMP DEFAULT NOW()  
);
```

The schema normalizes addresses into a separate table to save space and enable labeling. Events table stores both raw and decoded data. Specialized tables for ERC20 and ERC721 transfers enable fast token-specific queries. The ownership table maintains current state for instant owner lookups.

SECTION 5: DATABASE ACCESS LAYER

The database layer handles connection pooling, transactions, and address normalization.

```
import asyncpg  
from typing import Optional, Dict, List, Any  
from datetime import datetime
```

```
class IndexerDatabase:  
    def __init__(self, connection_string: str):  
        self.connection_string = connection_string  
        self.pool: Optional[asyncpg.Pool] = None  
        self.address_cache: Dict[str, int] = {}
```

```
    async def connect(self):  
        self.pool = await asyncpg.create_pool(  
            self.connection_string,  
            min_size=5,  
            max_size=20,
```

```
command_timeout = 60
)
```

```
async def close(self):
    if self.pool:
        await self.pool.close()
```

```
async def get_or_create_address(self, address: str, conn = None) ->
int:
    address = address.lower()
```

```
    if address in self.address_cache:
        return self.address_cache[address]
```

```
    use_conn = conn or await self.pool.acquire()
```

```
    try:
        row = await use_conn.fetchrow(
            "SELECT id FROM addresses WHERE address = $1",
            address
        )
```

```
    if row:
        address_id = row["id"]
    else:
        address_id = await use_conn.fetchval(
            """
```

```
INSERT INTO addresses (address)
VALUES ($1)
ON CONFLICT (address) DO UPDATE SET address =
EXCLUDED.address
RETURNING id
            """
        ,
        address
    )
```

```
    self.address_cache[address] = address_id
    return address_id
```

```
finally:
```

```
if not conn:
    await self.pool.release(use_conn)
```

```
async def insert_block(self, block: Dict):
    await self.pool.execute(
        """
```

```
INSERT INTO blocks (
    block_number, block_hash, parent_hash, timestamp,
    transaction_count, gas_used, gas_limit, base_fee_per_gas
) VALUES ($1, $2, $3, $4, $5, $6, $7, $8)
ON CONFLICT (block_number) DO NOTHING
        """,
        block["number"],
        block["hash"],
        block["parent_hash"],
        datetime.fromtimestamp(block["timestamp"]),
        block["transaction_count"],
        block.get("gas_used"),
        block.get("gas_limit"),
        block.get("base_fee_per_gas")
    )
```

```
async def insert_event(self, event: DecodedEvent):
    async with self.pool.acquire() as conn:
        async with conn.transaction():
            contract_id = await self.get_or_create_address(
                event.contract_address, conn
            )
```

```
        await conn.execute(
            """
INSERT INTO events (
    block_number, transaction_hash, log_index,
    contract_address_id, event_name,
    topic0, topic1, topic2, topic3,
    data, decoded_args
) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11)
ON CONFLICT (block_number, transaction_hash, log_index) DO
NOTHING
            """,
```

```

event.block_number,
event.transaction_hash,
event.log_index,
contract_id,
event.event_name,
event.raw_topics[0] if len(event.raw_topics) > 0 else None,
event.raw_topics[1] if len(event.raw_topics) > 1 else None,
event.raw_topics[2] if len(event.raw_topics) > 2 else None,
event.raw_topics[3] if len(event.raw_topics) > 3 else None,
event.raw_data,
json.dumps(event.args)
)

```

```

async def insert_erc20_transfer(self, event: DecodedEvent):
    async with self.pool.acquire() as conn:
        async with conn.transaction():
            token_id = await self.get_or_create_address(
                event.contract_address, conn
            )
            from_id = await self.get_or_create_address(
                event.args["from"], conn
            )
            to_id = await self.get_or_create_address(
                event.args["to"], conn
            )

```

```

        await conn.execute(
            """
INSERT INTO erc20_transfers (
    block_number, block_timestamp, transaction_hash, log_index,
    token_address_id, from_address_id, to_address_id, value
) VALUES ($1, $2, $3, $4, $5, $6, $7, $8)
ON CONFLICT (block_number, transaction_hash, log_index) DO
NOTHING
""",
            event.block_number,
            datetime.fromtimestamp(event.block_timestamp),
            event.transaction_hash,
            event.log_index,
            token_id,

```



```
"""
```

```
INSERT INTO nft_ownership (  
token_address_id, token_id, owner_address_id,  
acquired_block, acquired_timestamp  
) VALUES ($1, $2, $3, $4, $5)  
ON CONFLICT (token_address_id, token_id)  
DO UPDATE SET  
owner_address_id = EXCLUDED.owner_address_id,  
acquired_block = EXCLUDED.acquired_block,  
acquired_timestamp = EXCLUDED.acquired_timestamp
```

```
"""
```

```
,  
token_id,  
str(event.args["tokenId"]),  
to_id,  
event.block_number,  
datetime.fromtimestamp(event.block_timestamp)  
)
```

```
else:
```

```
await conn.execute(  
"""
```

```
"""
```

```
UPDATE nft_ownership  
SET owner_address_id = $1,  
acquired_block = $2,  
acquired_timestamp = $3  
WHERE token_address_id = $4 AND token_id = $5
```

```
"""
```

```
,  
to_id,  
event.block_number,  
datetime.fromtimestamp(event.block_timestamp),  
token_id,  
str(event.args["tokenId"])  
)
```

```
async def delete_events_from_block(self, from_block: int):  
    async with self.pool.acquire() as conn:  
        async with conn.transaction():  
            await conn.execute(  
                "DELETE FROM events WHERE block_number >= $1",  
                from_block  
            )
```

```

await conn.execute(
"DELETE FROM erc20_transfers WHERE block_number >= $1",
from_block
)
await conn.execute(
"DELETE FROM erc721_transfers WHERE block_number >= $1",
from_block
)
await conn.execute(
"DELETE FROM blocks WHERE block_number >= $1",
from_block
)

```

```

async def get_checkpoint(self) -> Optional[Dict]:
row = await self.pool.fetchrow(
"""
SELECT last_indexed_block, last_confirmed_block, updated_at
FROM indexer_checkpoints
WHERE indexer_name = 'main'
""")
)

```

```

if row:
return dict(row)
return None

```

```

async def save_checkpoint(self, checkpoint: Dict):
await self.pool.execute(
"""
INSERT INTO indexer_checkpoints (
indexer_name, last_indexed_block, last_confirmed_block, updated_at
) VALUES ('main', $1, $2, NOW())
ON CONFLICT (indexer_name) DO UPDATE SET
last_indexed_block = EXCLUDED.last_indexed_block,
last_confirmed_block = EXCLUDED.last_confirmed_block,
updated_at = NOW()
""",
,
checkpoint["last_indexed_block"],
checkpoint["last_confirmed_block"]
)

```


The database layer uses connection pooling for efficiency, transactions for atomicity, and address caching to reduce lookups. The NFT ownership table is automatically updated on transfers.

SECTION 6: REAL-TIME STREAMING

For applications needing instant updates, combine historical indexing with real-time event streaming.

```
class RealtimeEventStream:
    def __init__(
        self,
        ws_client: WebSocketRPCClient,
        database: IndexerDatabase,
        event_decoder: EventDecoder
    ):
        self.ws = ws_client
        self.db = database
        self.decoder = event_decoder
        self.subscribers: Dict[str, List[Callable]] = {}
        self.pending_events: Dict[str, DecodedEvent] = {}

    async def start(self, contract_addresses: List[str]):
        await self.ws.connect()

        for address in contract_addresses:
            await self.ws.subscribe_logs(
                self._handle_log,
                address=address
            )

        await self.ws.subscribe_new_blocks(self._handle_new_block)

    def subscribe(self, event_name: str, callback: Callable):
        if event_name not in self.subscribers:
            self.subscribers[event_name] = []
        self.subscribers[event_name].append(callback)
```

```
async def _handle_log(self, log: Dict):
    decoded = self.decoder.decode_log(log)
```

```
    if not decoded:
        return
```

```
    block_number = int(log["blockNumber"], 16)
```

```
    event = DecodedEvent(
        block_number=block_number,
        block_timestamp=0,
        transaction_hash=log["transactionHash"],
        log_index=int(log["logIndex"], 16),
        contract_address=log["address"].lower(),
        event_name=decoded["name"],
        args=decoded["args"],
        raw_data=log["data"],
        raw_topics=log["topics"]
    )
```

```
    event_key = f"{event.transaction_hash}:{event.log_index}"
    self.pending_events[event_key] = event
```

```
    await self._notify_subscribers(event, confirmed=False)
```

```
async def _handle_new_block(self, block_header: Dict):
    block_number = int(block_header["number"], 16)
    block_timestamp = int(block_header["timestamp"], 16)
    confirmations_needed = 12
    confirmed_block = block_number - confirmations_needed
```

```
    confirmed_events = []
    for key, event in list(self.pending_events.items()):
        if event.block_number <= confirmed_block:
            event.block_timestamp = block_timestamp
            confirmed_events.append(event)
            del self.pending_events[key]
```

```
    for event in confirmed_events:
        await self.db.insert_event(event)
```

```
await self._notify_subscribers(event, confirmed=True)
```

```
async def _notify_subscribers(self, event: DecodedEvent, confirmed:
bool):
```

```
    callbacks = self.subscribers.get(event.event_name, [])
```

```
    callbacks.extend(self.subscribers.get("*", []))
```

```
    notification = {
        "event": event,
        "confirmed": confirmed
    }
```

```
    for callback in callbacks:
```

```
        try:
```

```
            if asyncio.iscoroutinefunction(callback):
```

```
                await callback(notification)
```

```
            else:
```

```
                callback(notification)
```

```
        except Exception as e:
```

```
            print(f'Subscriber error: {e}')
```

```
class HybridIndexer:
```

```
    def __init__(
```

```
        self,
```

```
        historical_indexer: BlockchainIndexer,
```

```
        realtime_stream: RealtimeEventStream
```

```
    ):

```

```
        self.historical = historical_indexer
```

```
        self.realtime = realtime_stream
```

```
        self.caught_up = False
```

```
    async def start(self):
```

```
        historical_task = asyncio.create_task(self._run_historical())
```

```
        realtime_task = asyncio.create_task(self._run_realtime())
```

```
        await asyncio.gather(historical_task, realtime_task)
```

```
    async def _run_historical(self):
```

```
        while not self.caught_up:
```

```

current_block = await self.historical._get_current_block()

if self.historical.state.last_indexed_block >= current_block - 100:
    self.caught_up = True
    print("Historical indexer caught up, switching to realtime")
    break

await self.historical._index_batch()
await self.historical._process_confirmations()

while True:
    await asyncio.sleep(60)
    await self.historical._check_reorgs()
    await self.historical._save_checkpoint()

async def _run_realtime(self):
    while not self.caught_up:
        await asyncio.sleep(1)

contracts = list(self.historical.event_decoders.keys())
await self.realtime.start(contracts)

```

The hybrid indexer runs historical indexing until caught up, then switches to realtime streaming for instant updates. This provides both complete historical data and immediate event notifications.

SECTION 7: BATCH PROCESSING PATTERNS

Large-scale indexing requires efficient batch processing. Here are patterns for high-throughput data ingestion.

```

class BatchProcessor:
    def __init__(
        self,
        database: IndexerDatabase,
        batch_size: int = 1000,
        flush_interval: float = 5.0
    ):
        self.db = database

```

```
self.batch_size = batch_size
self.flush_interval = flush_interval
self.event_buffer: List[DecodedEvent] = []
self.erc20_buffer: List[DecodedEvent] = []
self.erc721_buffer: List[DecodedEvent] = []
self.last_flush = datetime.now()
self.lock = asyncio.Lock()
```

```
async def add_event(self, event: DecodedEvent):
    async with self.lock:
        self.event_buffer.append(event)
```

```
if event.event_name == "Transfer":
    if "value" in event.args:
        self.erc20_buffer.append(event)
    elif "tokenId" in event.args:
        self.erc721_buffer.append(event)
```

```
if self._should_flush():
    await self._flush()
```

```
def _should_flush(self) -> bool:
    if len(self.event_buffer) >= self.batch_size:
        return True
```

```
elapsed = (datetime.now() - self.last_flush).total_seconds()
if elapsed >= self.flush_interval and self.event_buffer:
    return True
```

```
return False
```

```
async def _flush(self):
    if not self.event_buffer:
        return
```

```
events = self.event_buffer
erc20_events = self.erc20_buffer
erc721_events = self.erc721_buffer
```

```
self.event_buffer = []
```

```
self.erc20_buffer = []
self.erc721_buffer = []
self.last_flush = datetime.now()
```

```
await self._batch_insert_events(events)
await self._batch_insert_erc20(erc20_events)
await self._batch_insert_erc721(erc721_events)
```

```
async def _batch_insert_events(self, events: List[DecodedEvent]):
    if not events:
        return
```

```
    async with self.db.pool.acquire() as conn:
        async with conn.transaction():
            unique_addresses = set()
            for event in events:
                unique_addresses.add(event.contract_address)
```

```
    for address in unique_addresses:
        await self.db.get_or_create_address(address, conn)
```

```
    records = []
    for event in events:
        contract_id = self.db.address_cache.get(event.contract_address)
```

```
    records.append((
        event.block_number,
        event.transaction_hash,
        event.log_index,
        contract_id,
        event.event_name,
        event.raw_topics[0] if len(event.raw_topics) > 0 else None,
        event.raw_topics[1] if len(event.raw_topics) > 1 else None,
        event.raw_topics[2] if len(event.raw_topics) > 2 else None,
        event.raw_topics[3] if len(event.raw_topics) > 3 else None,
        event.raw_data,
        json.dumps(event.args)
    ))
```

```
    await conn.executemany(
```

```
"""
```

```
INSERT INTO events (  
    block_number, transaction_hash, log_index,  
    contract_address_id, event_name,  
    topic0, topic1, topic2, topic3,  
    data, decoded_args  
) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11)  
ON CONFLICT DO NOTHING  
""",  
    records  
)
```

```
async def _batch_insert_erc20(self, events: List[DecodedEvent]):  
    if not events:  
        return
```

```
    async with self.db.pool.acquire() as conn:  
        async with conn.transaction():  
            unique_addresses = set()  
            for event in events:  
                unique_addresses.add(event.contract_address)  
                unique_addresses.add(event.args["from"])  
                unique_addresses.add(event.args["to"])
```

```
    for address in unique_addresses:  
        await self.db.get_or_create_address(address, conn)
```

```
    records = []  
    for event in events:  
        records.append((  
            event.block_number,  
            datetime.fromtimestamp(event.block_timestamp),  
            event.transaction_hash,  
            event.log_index,  
            self.db.address_cache.get(event.contract_address),  
            self.db.address_cache.get(event.args["from"]),  
            self.db.address_cache.get(event.args["to"]),  
            str(event.args["value"])  
        ))
```

```
await conn.executemany(
    """
```

```
INSERT INTO erc20_transfers (
    block_number, block_timestamp, transaction_hash, log_index,
    token_address_id, from_address_id, to_address_id, value
) VALUES ($1, $2, $3, $4, $5, $6, $7, $8)
ON CONFLICT DO NOTHING
    """,
    records
)
```

```
async def _batch_insert_erc721(self, events: List[DecodedEvent]):
    if not events:
        return
```

```
    async with self.db.pool.acquire() as conn:
        async with conn.transaction():
            unique_addresses = set()
            for event in events:
                unique_addresses.add(event.contract_address)
                unique_addresses.add(event.args["from"])
                unique_addresses.add(event.args["to"])
```

```
    for address in unique_addresses:
        await self.db.get_or_create_address(address, conn)
```

```
    records = []
    for event in events:
        records.append((
            event.block_number,
            datetime.fromtimestamp(event.block_timestamp),
            event.transaction_hash,
            event.log_index,
            self.db.address_cache.get(event.contract_address),
            self.db.address_cache.get(event.args["from"]),
            self.db.address_cache.get(event.args["to"]),
            str(event.args["tokenId"])
        ))
```

```
    await conn.executemany(
```



```
"""
```

```
INSERT INTO erc721_transfers (  
    block_number, block_timestamp, transaction_hash, log_index,  
    token_address_id, from_address_id, to_address_id, token_id  
) VALUES ($1, $2, $3, $4, $5, $6, $7, $8)  
ON CONFLICT DO NOTHING  
""",  
records  
)
```

```
async def force_flush(self):  
    async with self.lock:  
        await self._flush()
```

Batch processing buffers events and inserts them in bulk. This dramatically improves throughput compared to individual inserts. The processor handles both time-based and size-based flushing.

SECTION 8: QUERY PATTERNS

After indexing, you need efficient query patterns for common use cases.

```
class IndexerQueries:  
    def __init__(self, pool: asyncpg.Pool):  
        self.pool = pool
```

```
    async def get_token_transfers(  
        self,  
        token_address: str,  
        from_block: int = None,  
        to_block: int = None,  
        limit: int = 100,  
        offset: int = 0  
    ) -> List[Dict]:
```

```
        query = """  
        SELECT  
        t.block_number,
```

```

t.block_timestamp,
t.transaction_hash,
f.address as from_address,
to_addr.address as to_address,
t.value
FROM erc20_transfers t
JOIN addresses token ON t.token_address_id = token.id
JOIN addresses f ON t.from_address_id = f.id
JOIN addresses to_addr ON t.to_address_id = to_addr.id
WHERE token.address = $1
"""

```

```

params = [token_address.lower()]
param_count = 1

```

```

if from_block:
    param_count += 1
    query += f" AND t.block_number >= ${param_count}"
    params.append(from_block)

```

```

if to_block:
    param_count += 1
    query += f" AND t.block_number <= ${param_count}"
    params.append(to_block)

```

```

query += f" ORDER BY t.block_number DESC, t.log_index DESC"
query += f" LIMIT ${param_count + 1} OFFSET ${param_count + 2}"
params.extend([limit, offset])

```

```

rows = await self.pool.fetch(query, *params)
return [dict(row) for row in rows]

```

```

async def get_address_activity(
    self,
    address: str,
    event_types: List[str] = None,
    limit: int = 100
) -> List[Dict]:

```

```

query = """
SELECT
e.block_number,
e.transaction_hash,
e.event_name,
e.decoded_args,
c.address as contract_address
FROM events e
JOIN addresses c ON e.contract_address_id = c.id
WHERE e.decoded_args @> $1
"""

```

```

search_pattern = json.dumps({"from": address.lower()})

```

```

params = [search_pattern]

```

```

if event_types:
query += " AND e.event_name = ANY($2)"
params.append(event_types)

```

```

query += " ORDER BY e.block_number DESC LIMIT $" +
str(len(params) + 1)
params.append(limit)

```

```

rows = await self.pool.fetch(query, *params)
return [dict(row) for row in rows]

```

```

async def get_nft_owner(
self,
contract_address: str,
token_id: str
) -> Optional[str]:

```

```

row = await self.pool.fetchrow(
"""

```

```

SELECT owner.address
FROM nft_ownership o
JOIN addresses token ON o.token_address_id = token.id
JOIN addresses owner ON o.owner_address_id = owner.id
WHERE token.address = $1 AND o.token_id = $2

```

```

"""
contract_address.lower(),
token_id
)

```

```

return row["address"] if row else None

```

```

async def get_collection_holders(
self,
contract_address: str,
limit: int = 100
) -> List[Dict]:

```

```

rows = await self.pool.fetch(
"""

```

```

SELECT
owner.address,
COUNT(*) as token_count,
array_agg(o.token_id) as token_ids
FROM nft_ownership o
JOIN addresses token ON o.token_address_id = token.id
JOIN addresses owner ON o.owner_address_id = owner.id
WHERE token.address = $1
GROUP BY owner.address
ORDER BY COUNT(*) DESC
LIMIT $2
"""
,
contract_address.lower(),
limit
)

```

```

return [dict(row) for row in rows]

```

```

async def get_transfer_volume(
self,
token_address: str,
start_time: datetime,
end_time: datetime,
interval: str = "1 day"
) -> List[Dict]:

```

```

rows = await self.pool.fetch(
    """
SELECT
date_trunc($4, block_timestamp) as period,
COUNT(*) as transfer_count,
COUNT(DISTINCT from_address_id) as unique_senders,
COUNT(DISTINCT to_address_id) as unique_receivers,
SUM(value::numeric) as total_volume
FROM erc20_transfers t
JOIN addresses token ON t.token_address_id = token.id
WHERE token.address = $1
AND block_timestamp >= $2
AND block_timestamp < $3
GROUP BY period
ORDER BY period
    """,
    token_address.lower(),
    start_time,
    end_time,
    interval.split()[1]
)

```

```

return [dict(row) for row in rows]

```

```

async def search_events(
    self,
    contract_address: str = None,
    event_name: str = None,
    args_filter: Dict = None,
    from_block: int = None,
    to_block: int = None,
    limit: int = 100
) -> List[Dict]:

```

```

conditions = []
params = []
param_count = 0

```

```

if contract_address:

```

```
param_count += 1
conditions.append(f'c.address = ${param_count}')
params.append(contract_address.lower())
```

```
if event_name:
    param_count += 1
    conditions.append(f'e.event_name = ${param_count}')
    params.append(event_name)
```

```
if args_filter:
    param_count += 1
    conditions.append(f'e.decoded_args @> ${param_count}')
    params.append(json.dumps(args_filter))
```

```
if from_block:
    param_count += 1
    conditions.append(f'e.block_number >= ${param_count}')
    params.append(from_block)
```

```
if to_block:
    param_count += 1
    conditions.append(f'e.block_number <= ${param_count}')
    params.append(to_block)
```

```
where_clause = " AND ".join(conditions) if conditions else "1 = 1"
```

```
param_count += 1
params.append(limit)
```

```
query = f"""
SELECT
e.block_number,
e.transaction_hash,
e.log_index,
e.event_name,
e.decoded_args,
c.address as contract_address
FROM events e
JOIN addresses c ON e.contract_address_id = c.id
WHERE {where_clause}
```

```
ORDER BY e.block_number DESC, e.log_index DESC
LIMIT ${param_count}
""""
```

```
rows = await self.pool.fetch(query, *params)
return [dict(row) for row in rows]
```

The query layer provides common access patterns with proper parameterization and indexing hints. JSONB queries enable flexible searching of decoded event arguments.

SECTION 9: PARTITIONING FOR SCALE

As data grows, partition tables by block number for better performance and easier maintenance.

```
CREATE TABLE events_partitioned (
    id BIGSERIAL,
    block_number BIGINT NOT NULL,
    transaction_hash VARCHAR(66) NOT NULL,
    log_index INTEGER NOT NULL,
    contract_address_id INTEGER,
    event_name VARCHAR(100) NOT NULL,
    topic0 VARCHAR(66),
    topic1 VARCHAR(66),
    topic2 VARCHAR(66),
    topic3 VARCHAR(66),
    data TEXT,
    decoded_args JSONB,
    indexed_at TIMESTAMP DEFAULT NOW(),
```

```
PRIMARY KEY (block_number, id)
) PARTITION BY RANGE (block_number);
```

```
CREATE TABLE events_p_0_1m PARTITION OF events_partitioned
FOR VALUES FROM (0) TO (1000000);
```

```
CREATE TABLE events_p_1m_2m PARTITION OF events_partitioned
FOR VALUES FROM (1000000) TO (2000000);
```

```
CREATE TABLE events_p_2m_3m PARTITION OF events_partitioned
FOR VALUES FROM (2000000) TO (3000000);
```

```
CREATE OR REPLACE FUNCTION create_events_partition(
    start_block BIGINT,
    end_block BIGINT
) RETURNS void AS $$
DECLARE
    partition_name TEXT;
BEGIN
    partition_name := 'events_p_' || start_block || '_' || end_block;
```

```
EXECUTE format(
    'CREATE TABLE IF NOT EXISTS %I PARTITION OF
events_partitioned
FOR VALUES FROM (%s) TO (%s)',
    partition_name,
    start_block,
    end_block
);
```

```
EXECUTE format(
    'CREATE INDEX IF NOT EXISTS %I ON %I (contract_address_id)',
    'idx_' || partition_name || '_contract',
    partition_name
);
```

```
EXECUTE format(
    'CREATE INDEX IF NOT EXISTS %I ON %I (event_name)',
    'idx_' || partition_name || '_name',
    partition_name
);
END;
$$ LANGUAGE plpgsql;
```

```
SELECT create_events_partition(3000000, 4000000);
SELECT create_events_partition(4000000, 5000000);
```

Partitioning enables fast queries on recent data, efficient deletion of

old partitions, parallel query execution, and smaller index sizes per partition.

SECTION 10: COMPLETE INDEXER DEPLOYMENT

Here is a complete Docker Compose deployment for a production indexer.

version: '3.8'

services:

postgres:

image: postgres:15

container_name: indexer_db

restart: unless-stopped

environment:

POSTGRES_USER: indexer

POSTGRES_PASSWORD: \${DB_PASSWORD}

POSTGRES_DB: blockchain_index

volumes:

- postgres_data:/var/lib/postgresql/data

- ./init.sql:/docker-entrypoint-initdb.d/init.sql

ports:

- "5432:5432"

healthcheck:

test: ["CMD-SHELL", "pg_isready -U indexer"]

interval: 10s

timeout: 5s

retries: 5

redis:

image: redis:7-alpine

container_name: indexer_cache

restart: unless-stopped

volumes:

- redis_data:/data

ports:

- "6379:6379"

healthcheck:

test: ["CMD", "redis-cli", "ping"]
interval: 10s
timeout: 5s
retries: 5

indexer:
build: ./indexer
container_name: blockchain_indexer
restart: unless-stopped
depends_on:
postgres:
condition: service_healthy
redis:
condition: service_healthy
environment:
DATABASE_URL: postgresql://indexer:\${DB_PASSWORD}
@postgres:5432/blockchain_index
REDIS_URL: redis://redis:6379
RPC_URL_PRIMARY: \${RPC_URL_PRIMARY}
RPC_URL_SECONDARY: \${RPC_URL_SECONDARY}
WS_URL: \${WS_URL}
START_BLOCK: \${START_BLOCK:-0}
BATCH_SIZE: \${BATCH_SIZE:-100}
CONFIRMATIONS: \${CONFIRMATIONS:-12}
volumes:
- ./config:/app/config
healthcheck:
test: ["CMD", "python", "-c", "import requests; requests.get('http://localhost:8080/health')"]
interval: 30s
timeout: 10s
retries: 3

api:
build: ./api
container_name: indexer_api
restart: unless-stopped
depends_on:
postgres:
condition: service_healthy

```
ports:
- "3000:3000"
environment:
DATABASE_URL: postgresql://indexer:${DB_PASSWORD}
@postgres:5432/blockchain_index
REDIS_URL: redis://redis:6379
healthcheck:
test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
interval: 30s
timeout: 10s
retries: 3

volumes:
postgres_data:
redis_data:
```

The Dockerfile for the indexer.

FROM python:3.11-slim

WORKDIR /app

```
RUN apt-get update && apt-get install -y \
build-essential \
&& rm -rf /var/lib/apt/lists/*
```

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "-m", "indexer.main"]

The main entry point.

```
import asyncio
import os
import signal
from indexer.core import BlockchainIndexer, IndexerConfig
from indexer.database import IndexerDatabase
```

```

from indexer.rpc import ResilientRPCClient

async def main():
    providers = [
        {
            "name": "primary",
            "url": os.environ["RPC_URL_PRIMARY"],
            "priority": 0
        },
        {
            "name": "secondary",
            "url": os.environ["RPC_URL_SECONDARY"],
            "priority": 1
        }
    ]

    rpc_client = ResilientRPCClient(providers)

    database = IndexerDatabase(os.environ["DATABASE_URL"])
    await database.connect()

    config = IndexerConfig(
        start_block=int(os.environ.get("START_BLOCK", 0)),
        batch_size=int(os.environ.get("BATCH_SIZE", 100)),
        confirmations=int(os.environ.get("CONFIRMATIONS", 12)),
        contracts={}
    )

    indexer = BlockchainIndexer(rpc_client, database, config)

    indexer.register_contract(
        "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
        ERC20_ABI
    )

    async def handle_transfer(event):
        if "value" in event.args:
            await database.insert_erc20_transfer(event)
        elif "tokenId" in event.args:
            await database.insert_erc721_transfer(event)

```

```

indexer.register_handler("Transfer", handle_transfer)

shutdown_event = asyncio.Event()

def shutdown_handler(sig):
    print(f"Received signal {sig}, shutting down...")
    shutdown_event.set()

loop = asyncio.get_running_loop()
for sig in (signal.SIGTERM, signal.SIGINT):
    loop.add_signal_handler(sig, lambda s = sig: shutdown_handler(s))

indexer_task = asyncio.create_task(indexer.start())

await shutdown_event.wait()

await indexer.stop()
await database.close()
await rpc_client.close()

if __name__ == "__main__":
    asyncio.run(main())

```

This chapter covered the complete data pipeline from raw blockchain data to a queryable database. You learned indexer architecture with block fetching and event decoding, event decoding with ABI parsing, database schema design for blockchain data, batch processing for high throughput, real-time streaming with WebSockets, query patterns for common use cases, partitioning for scale, and production deployment with Docker. The indexed data serves as the foundation for all application features in the following chapters.

CHAPTER 3: CACHING STRATEGIES

Blockchain data presents a unique caching challenge. Users expect millisecond responses but block times are 12 seconds on Ethereum and confirmations take minutes. RPC calls are expensive and rate-limited. Without proper caching, your application will be slow, expensive, and unreliable. This chapter teaches multi-layer caching

strategies specifically designed for blockchain data.

SECTION 1: THE BLOCKCHAIN CACHING PROBLEM

Blockchain data has distinct characteristics that affect caching strategy.

Immutability is both friend and enemy. Once confirmed, historical data never changes. A transaction from block 1000000 will always have the same data. This makes historical data infinitely cacheable. But recent data is uncertain due to chain reorganizations.

Freshness requirements vary wildly. Token balances need to be accurate within the last block. Historical transfers can be cached forever. Price data might need real-time updates while metadata can be cached for days.

RPC costs accumulate quickly. Each call costs money or consumes rate limit budget. Redundant calls waste resources. Caching is not optional for production applications.

Here is a multi-layer caching architecture.

The first layer is application memory. Sub-millisecond access for hot data. Limited size. Lost on restart.

The second layer is Redis. Millisecond access for warm data. Shared across application instances. Survives restarts.

The third layer is the database. Still faster than RPC. Contains indexed historical data.

The fourth layer is RPC. The source of truth. Slowest and most expensive.

Requests flow down through layers. Data flows up to populate caches.

SECTION 2: IN-MEMORY CACHING

The fastest cache lives in your application process. Use it for frequently accessed, small data.

```
import time
from typing import Any, Optional, Dict, Callable
from dataclasses import dataclass
from collections import OrderedDict
import threading
import asyncio

@dataclass
class CacheEntry:
    value: Any
    expires_at: float
    block_number: Optional[int] = None

class LRUCache:
    def __init__(self, max_size: int = 1000):
        self.max_size = max_size
        self.cache: OrderedDict[str, CacheEntry] = OrderedDict()
        self.lock = threading.RLock()

    def get(self, key: str) -> Optional[Any]:
        with self.lock:
            if key not in self.cache:
                return None

            entry = self.cache[key]

            if entry.expires_at and time.time() > entry.expires_at:
                del self.cache[key]
                return None

            self.cache.move_to_end(key)
            return entry.value

    def set(
        self,
```

```

key: str,
value: Any,
ttl_seconds: float = None,
block_number: int = None
):
    with self.lock:
        expires_at = time.time() + ttl_seconds if ttl_seconds else None

        self.cache[key] = CacheEntry(
            value=value,
            expires_at=expires_at,
            block_number=block_number
        )

    self.cache.move_to_end(key)

    while len(self.cache) > self.max_size:
        self.cache.popitem(last=False)

def delete(self, key: str):
    with self.lock:
        self.cache.pop(key, None)

def invalidate_before_block(self, block_number: int):
    with self.lock:
        keys_to_delete = []

        for key, entry in self.cache.items():
            if entry.block_number and entry.block_number < block_number:
                keys_to_delete.append(key)

        for key in keys_to_delete:
            del self.cache[key]

def clear(self):
    with self.lock:
        self.cache.clear()

def stats(self) -> Dict[str, int]:
    with self.lock:

```



```
return {  
    "size": len(self.cache),  
    "max_size": self.max_size  
}
```

```
class TTLCache:  
    def __init__(self):  
        self.cache: Dict[str, CacheEntry] = {}  
        self.lock = threading.RLock()  
  
    def get(self, key: str) -> Optional[Any]:  
        with self.lock:  
            if key not in self.cache:  
                return None  
  
            entry = self.cache[key]  
  
            if time.time() > entry.expires_at:  
                del self.cache[key]  
                return None  
  
            return entry.value  
  
    def set(self, key: str, value: Any, ttl_seconds: float):  
        with self.lock:  
            self.cache[key] = CacheEntry(  
                value=value,  
                expires_at=time.time() + ttl_seconds  
            )  
  
    def delete(self, key: str):  
        with self.lock:  
            self.cache.pop(key, None)  
  
    def cleanup_expired(self):  
        with self.lock:  
            now = time.time()  
            keys_to_delete = [  
                key for key, entry in self.cache.items()
```

```
if now > entry.expires_at  
]  
for key in keys_to_delete:  
del self.cache[key]
```

```
class BlockAwareCache:  
    def __init__(self, confirmations: int = 12):  
        self.confirmed_cache = LRUCache(max_size=10000)  
        self.pending_cache = TTLCache()  
        self.confirmations = confirmations  
        self.current_block = 0  
  
    def set_current_block(self, block_number: int):  
        old_block = self.current_block  
        self.current_block = block_number  
  
    if block_number > old_block:  
        confirmed_threshold = block_number - self.confirmations  
        self.confirmed_cache.invalidate_before_block(confirmed_threshold)  
  
    def get(self, key: str, min_block: int = None) -> Optional[Any]:  
        confirmed = self.confirmed_cache.get(key)  
  
        if confirmed is not None:  
            return confirmed  
  
        return self.pending_cache.get(key)  
  
    def set_confirmed(self, key: str, value: Any, block_number: int):  
        self.confirmed_cache.set(key, value, block_number=block_number)  
  
    def set_pending(self, key: str, value: Any, ttl_seconds: float = 12.0):  
        self.pending_cache.set(key, value, ttl_seconds)
```

The block-aware cache distinguishes between confirmed and pending data. Confirmed data is cached indefinitely. Pending data expires after one block time.

SECTION 3: REDIS CACHING LAYER

Redis provides shared caching across application instances with persistence and advanced data structures.

```
import redis.asyncio as redis
import json
from typing import Any, Optional, List, Dict, Set
import hashlib

class RedisCache:
    def __init__(self, redis_url: str, key_prefix: str = "bc"):
        self.redis = redis.from_url(redis_url)
        self.prefix = key_prefix
        self.stats = {"hits": 0, "misses": 0}

    def _key(self, key: str) -> str:
        return f"{self.prefix}:{key}"

    async def get(self, key: str) -> Optional[Any]:
        raw = await self.redis.get(self._key(key))

        if raw is None:
            self.stats["misses"] += 1
            return None

        self.stats["hits"] += 1
        return json.loads(raw)

    async def set(
        self,
        key: str,
        value: Any,
        ttl_seconds: int = None,
        block_number: int = None
    ):
        data = {
            "value": value,
            "block": block_number,
            "cached_at": time.time()
```

```
}
```

```
await self.redis.set(
    self._key(key),
    json.dumps(data),
    ex = ttl_seconds
)
```

```
async def delete(self, key: str):
    await self.redis.delete(self._key(key))
```

```
async def delete_pattern(self, pattern: str):
    cursor = 0
```

```
while True:
    cursor, keys = await self.redis.scan(
        cursor,
        match=self._key(pattern),
        count=100
    )
```

```
if keys:
    await self.redis.delete(*keys)
```

```
if cursor == 0:
    break
```

```
async def get_many(self, keys: List[str]) -> Dict[str, Any]:
    if not keys:
        return {}
```

```
prefixed_keys = [self._key(k) for k in keys]
values = await self.redis.mget(prefixed_keys)
```

```
result = {}
for key, value in zip(keys, values):
    if value is not None:
        result[key] = json.loads(value)
        self.stats["hits"] += 1
    else:
```

```

self.stats["misses"] += 1

return result

async def set_many(
    self,
    items: Dict[str, Any],
    ttl_seconds: int = None
):
    pipe = self.redis.pipeline()

    for key, value in items.items():
        data = json.dumps({"value": value, "cached_at": time.time()})

        if ttl_seconds:
            pipe.setex(self._key(key), ttl_seconds, data)
        else:
            pipe.set(self._key(key), data)

    await pipe.execute()

    async def increment(self, key: str, amount: int = 1) -> int:
        return await self.redis.incrby(self._key(key), amount)

    async def get_hit_rate(self) -> float:
        total = self.stats["hits"] + self.stats["misses"]
        if total == 0:
            return 0.0
        return self.stats["hits"] / total

    async def close(self):
        await self.redis.close()

class BlockchainRedisCache(RedisCache):
    def __init__(self, redis_url: str):
        super().__init__(redis_url, "chain")

    async def cache_balance(
        self,

```

```
address: str,  
token_address: str,  
balance: str,  
block_number: int  
):  
key = f"balance:{token_address}:{address}"  
await self.set(key, balance, ttl_seconds = 30,  
block_number = block_number)
```

```
async def get_balance(  
self,  
address: str,  
token_address: str  
) -> Optional[str]:  
key = f"balance:{token_address}:{address}"  
data = await self.get(key)  
return data["value"] if data else None
```

```
async def cache_nft_owner(  
self,  
contract_address: str,  
token_id: str,  
owner: str,  
block_number: int  
):  
key = f"nft_owner:{contract_address}:{token_id}"  
await self.set(key, owner, ttl_seconds = 60,  
block_number = block_number)
```

```
async def cache_metadata(  
self,  
contract_address: str,  
token_id: str,  
metadata: Dict  
):  
key = f"metadata:{contract_address}:{token_id}"  
await self.set(key, metadata, ttl_seconds = 86400)
```

```
async def cache_contract_info(  
self,
```

```

contract_address: str,
info: Dict
):
key = f'contract:{contract_address}'
await self.set(key, info, ttl_seconds = 3600)

async def cache_block(self, block_number: int, block_data: Dict):
key = f'block:{block_number}'
await self.set(key, block_data)

async def get_block(self, block_number: int) -> Optional[Dict]:
key = f'block:{block_number}'
data = await self.get(key)
return data["value"] if data else None

async def cache_transaction(self, tx_hash: str, tx_data: Dict):
key = f'tx:{tx_hash}'
await self.set(key, tx_data)

async def cache_gas_price(self, gas_price: int, ttl_seconds: int = 10):
await self.set("gas_price", gas_price, ttl_seconds = ttl_seconds)

async def get_gas_price(self) -> Optional[int]:
data = await self.get("gas_price")
return data["value"] if data else None

```

Redis caching uses structured keys for different data types. Balances have short TTLs because they change frequently. Metadata has long TTLs because it rarely changes.

SECTION 4: CACHE INVALIDATION STRATEGIES

Cache invalidation is the hardest problem in caching. For blockchain data, you have several strategies.

Time-based invalidation expires data after a fixed time. Simple but may serve stale data.

Block-based invalidation expires data after a certain number of

blocks. Aligns with blockchain time.

Event-based invalidation listens for blockchain events and invalidates affected cache entries.

```
class CacheInvalidator:
    def __init__(
        self,
        memory_cache: BlockAwareCache,
        redis_cache: BlockchainRedisCache
    ):
        self.memory = memory_cache
        self.redis = redis_cache
        self.invalidation_rules: Dict[str, List[str]] = {}

    def register_invalidation_rule(
        self,
        event_name: str,
        cache_patterns: List[str]
    ):
        self.invalidation_rules[event_name] = cache_patterns

    async def handle_event(self, event: Dict):
        event_name = event.get("event_name")

        if event_name not in self.invalidation_rules:
            return

        patterns = self.invalidation_rules[event_name]

        for pattern in patterns:
            cache_keys = self._expand_pattern(pattern, event)

            for key in cache_keys:
                self.memory.pending_cache.delete(key)
                await self.redis.delete(key)

    def _expand_pattern(self, pattern: str, event: Dict) -> List[str]:
        args = event.get("args", {})
```



```
if "{" not in pattern:  
    return [pattern]
```

```
keys = []
```

```
for arg_name, arg_value in args.items():  
    placeholder = "{" + arg_name + "}"
```

```
if placeholder in pattern:  
    expanded = pattern.replace(placeholder, str(arg_value).lower())  
    keys.append(expanded)
```

```
return keys if keys else [pattern]
```

```
async def handle_new_block(self, block_number: int):  
    self.memory.set_current_block(block_number)
```

```
    await self.redis.delete("gas_price")
```

```
async def handle_reorg(self, from_block: int):  
    self.memory.confirmed_cache.invalidate_before_block(from_block)
```

```
    await self.redis.delete_pattern(f"block:{from_block}*")  
    await self.redis.delete_pattern("balance:*")  
    await self.redis.delete_pattern("nft_owner:*")
```

```
class TransferInvalidator(CacheInvalidator):  
    def __init__(self, memory_cache, redis_cache):  
        super().__init__(memory_cache, redis_cache)
```

```
    self.register_invalidation_rule("Transfer", [  
        "balance:{contract}:{from}",  
        "balance:{contract}:{to}",  
        "nft_owner:{contract}:{tokenId}"  
    ])
```

```
    async def handle_transfer(self, event: Dict):  
        contract = event["contract_address"]  
        args = event["args"]
```

```

from_addr = args.get("from", "").lower()
to_addr = args.get("to", "").lower()

await self.redis.delete(f"balance:{contract}:{from_addr}")
await self.redis.delete(f"balance:{contract}:{to_addr}")

if "tokenId" in args:
    token_id = args["tokenId"]
    await self.redis.delete(f"nft_owner:{contract}:{token_id}")

```

Event-based invalidation is the most accurate. When a Transfer event occurs, invalidate the balance caches for both sender and recipient.

SECTION 5: TIERED CACHING IMPLEMENTATION

Combine all cache layers into a unified interface.

```

class TieredCache:
    def __init__(
        self,
        memory_cache: BlockAwareCache,
        redis_cache: RedisCache,
        rpc_client,
        database
    ):
        self.memory = memory_cache
        self.redis = redis_cache
        self.rpc = rpc_client
        self.db = database
        self.stats = {
            "memory_hits": 0,
            "redis_hits": 0,
            "db_hits": 0,
            "rpc_calls": 0
        }

```

```

async def get_balance(

```

```
self,  
address: str,  
token_address: str = None,  
block_number: str = "latest"  
) -> int:
```

```
cache_key = f'balance:{token_address or 'eth'}:{address.lower()}'
```

```
memory_value = self.memory.get(cache_key)  
if memory_value is not None:  
self.stats["memory_hits"] += 1  
return memory_value
```

```
redis_data = await self.redis.get(cache_key)  
if redis_data is not None:  
self.stats["redis_hits"] += 1  
value = redis_data["value"]  
self.memory.set_pending(cache_key, value)  
return value
```

```
if token_address:  
balance = await self._fetch_token_balance(address, token_address,  
block_number)  
else:  
balance = await self._fetch_eth_balance(address, block_number)  
  
self.stats["rpc_calls"] += 1
```

```
self.memory.set_pending(cache_key, balance, ttl_seconds=12)  
await self.redis.set(cache_key, balance, ttl_seconds=30)
```

```
return balance
```

```
async def _fetch_eth_balance(self, address: str, block: str) -> int:  
result = await self.rpc.call("eth_getBalance", [address, block])  
return int(result, 16)
```

```
async def _fetch_token_balance(  
self,  
address: str,
```

```
token_address: str,  
block: str  
) -> int:
```

```
balance_of_selector = "0x70a08231"  
padded_address = address[2:].lower().zfill(64)  
data = balance_of_selector + padded_address
```

```
result = await self.rpc.call("eth_call", [  
{"to": token_address, "data": data},  
block  
)
```

```
return int(result, 16) if result and result != "0x" else 0
```

```
async def get_nft_owner(  
self,  
contract_address: str,  
token_id: str  
) -> Optional[str]:
```

```
cache_key = f'nft_owner:{contract_address.lower()}:{token_id}'
```

```
memory_value = self.memory.get(cache_key)  
if memory_value is not None:  
self.stats["memory_hits"] += 1  
return memory_value
```

```
redis_data = await self.redis.get(cache_key)  
if redis_data is not None:  
self.stats["redis_hits"] += 1  
return redis_data["value"]
```

```
db_owner = await self.db.get_nft_owner(contract_address, token_id)  
if db_owner:  
self.stats["db_hits"] += 1  
self.memory.set_confirmed(cache_key, db_owner, block_number=0)  
await self.redis.set(cache_key, db_owner, ttl_seconds=300)  
return db_owner
```

```
owner = await self._fetch_nft_owner(contract_address, token_id)
self.stats["rpc_calls"] += 1
```

```
if owner:
    self.memory.set_pending(cache_key, owner)
    await self.redis.set(cache_key, owner, ttl_seconds=60)
```

```
return owner
```

```
async def _fetch_nft_owner(
    self,
    contract_address: str,
    token_id: str
) -> Optional[str]:
```

```
owner_of_selector = "0x6352211e"
padded_token_id = hex(int(token_id))[2:].zfill(64)
data = owner_of_selector + padded_token_id
```

```
try:
    result = await self.rpc.call("eth_call", [
        {"to": contract_address, "data": data},
        "latest"
    ])
```

```
if result and len(result) >= 66:
    return "0x" + result[26:66]
return None
```

```
except Exception:
    return None
```

```
async def get_block(self, block_number: int) -> Optional[Dict]:
    cache_key = f"block:{block_number}"
```

```
memory_value = self.memory.get(cache_key)
if memory_value is not None:
    self.stats["memory_hits"] += 1
    return memory_value
```

```
redis_data = await self.redis.get(cache_key)
if redis_data is not None:
    self.stats["redis_hits"] += 1
    block = redis_data["value"]
    self.memory.set_confirmed(cache_key, block,
block_number=block_number)
    return block
```

```
block = await self.rpc.call(
    "eth_getBlockByNumber",
    [hex(block_number), True]
)
self.stats["rpc_calls"] += 1
```

```
if block:
    self.memory.set_confirmed(cache_key, block,
block_number=block_number)
    await self.redis.set(cache_key, block)
```

```
return block
```

```
async def get_metadata(
    self,
    contract_address: str,
    token_id: str
) -> Optional[Dict]:
```

```
cache_key = f'metadata:{contract_address.lower()}:{token_id}'
```

```
memory_value = self.memory.get(cache_key)
if memory_value is not None:
    self.stats["memory_hits"] += 1
    return memory_value
```

```
redis_data = await self.redis.get(cache_key)
if redis_data is not None:
    self.stats["redis_hits"] += 1
    metadata = redis_data["value"]
    self.memory.set_confirmed(cache_key, metadata, block_number=0)
    return metadata
```

```
metadata = await self._fetch_metadata(contract_address, token_id)
self.stats["rpc_calls"] += 1
```

```
if metadata:
    self.memory.set_confirmed(cache_key, metadata, block_number=0)
    await self.redis.set(cache_key, metadata, ttl_seconds=86400)
```

```
return metadata
```

```
async def _fetch_metadata(
    self,
    contract_address: str,
    token_id: str
) -> Optional[Dict]:
```

```
token_uri_selector = "0xc87b56dd"
padded_token_id = hex(int(token_id))[2:].zfill(64)
data = token_uri_selector + padded_token_id
```

```
try:
    result = await self.rpc.call("eth_call", [
        {"to": contract_address, "data": data},
        "latest"
    ])
```

```
if not result or result == "0x":
    return None
```

```
uri = self._decode_string(result)
```

```
if uri.startswith("ipfs://"):
    uri = "https://ipfs.io/ipfs/" + uri[7:]
```

```
async with httpx.AsyncClient() as client:
    response = await client.get(uri, timeout=30)
    return response.json()
```

```
except Exception:
    return None
```

```

def _decode_string(self, hex_data: str) -> str:
    data = bytes.fromhex(hex_data[2:])

    if len(data) < 64:
        return ""

    offset = int.from_bytes(data[0:32], "big")
    length = int.from_bytes(data[offset:offset + 32], "big")
    string_data = data[offset + 32:offset + 32 + length]

    return string_data.decode("utf-8")

def get_stats(self) -> Dict:
    total = sum(self.stats.values())

    return {
        **self.stats,
        "total_requests": total,
        "memory_hit_rate": self.stats["memory_hits"] / total if total > 0 else
0,
        "cache_hit_rate": (
            self.stats["memory_hits"] + self.stats["redis_hits"] +
            self.stats["db_hits"]
        ) / total if total > 0 else 0
    }

```

The tiered cache checks each layer in order. If data is found in a lower layer, it populates higher layers for future requests.

SECTION 6: CACHE WARMING

Pre-populate caches with frequently accessed data to avoid cold-start latency.

```

class CacheWarmer:
    def __init__(
        self,
        tiered_cache: TieredCache,

```



```

database,
rpc_client
):
self.cache = tiered_cache
self.db = database
self.rpc = rpc_client

async def warm_recent_blocks(self, count: int = 100):
current_block = await self.rpc.call("eth_blockNumber")
current_block = int(current_block, 16)

start_block = current_block - count

batch_size = 10
for i in range(start_block, current_block + 1, batch_size):
tasks = []
for block_num in range(i, min(i + batch_size, current_block + 1)):
tasks.append(self.cache.get_block(block_num))

await asyncio.gather(*tasks)

print(f"Warmed {count} recent blocks")

async def warm_top_tokens(self, token_list: List[Dict]):
for token in token_list:
contract_address = token["address"]

contract_info = {
"address": contract_address,
"name": token.get("name"),
"symbol": token.get("symbol"),
"decimals": token.get("decimals")
}

await self.cache.redis.set(
f'contract:{contract_address.lower()}',
contract_info,
ttl_seconds = 3600
)

```

```

print(f"Warmed {len(token_list)} token contracts")

async def warm_active_addresses(self, addresses: List[str]):
    tasks = []

    for address in addresses:
        tasks.append(self.cache.get_balance(address))

    await asyncio.gather(*tasks, return_exceptions=True)
    print(f"Warmed balances for {len(addresses)} addresses")

    async def warm_nft_collection(
        self,
        contract_address: str,
        token_ids: List[str]
    ):
        batch_size = 50

        for i in range(0, len(token_ids), batch_size):
            batch = token_ids[i:i + batch_size]
            tasks = []

            for token_id in batch:
                tasks.append(
                    self.cache.get_metadata(contract_address, token_id)
                )

            await asyncio.gather(*tasks, return_exceptions=True)

        print(f"Warmed {len(token_ids)} NFT metadata entries")

    async def warm_from_database(self):
        top_contracts = await self.db.pool.fetch(
            """
            SELECT address, COUNT(*) as event_count
            FROM addresses a
            JOIN events e ON a.id = e.contract_address_id
            GROUP BY address
            ORDER BY event_count DESC
            LIMIT 100
        """
    )

```

```
"""
```

```
)
```

```
for row in top_contracts:
    await self.cache.redis.set(
        f'hot_contract:{row['address']}',
        True,
        ttl_seconds = 3600
    )
```

```
print(f'Marked {len(top_contracts)} hot contracts')
```

```
async def run_warming_schedule(self):
    while True:
        try:
            await self.warm_recent_blocks(50)
            await asyncio.sleep(60)

        except Exception as e:
            print(f'Cache warming error: {e}')
            await asyncio.sleep(30)
```

Cache warming runs on startup and periodically to keep hot data ready. Focus on data that is frequently accessed and expensive to fetch.

SECTION 7: CACHE COMPRESSION

Large cached objects consume memory and network bandwidth. Compression reduces both.

```
import zlib
import pickle
from typing import Any

class CompressedCache:
    def __init__(
        self,
        redis_cache: RedisCache,
```

```

compression_threshold: int = 1000
):
self.redis = redis_cache
self.threshold = compression_threshold

async def get(self, key: str) -> Optional[Any]:
raw = await self.redis.redis.get(self.redis._key(key))

if raw is None:
return None

if raw[:2] == b'\x78\x9c':
decompressed = zlib.decompress(raw)
return pickle.loads(decompressed)

return json.loads(raw)

async def set(
self,
key: str,
value: Any,
ttl_seconds: int = None
):
serialized = json.dumps(value)

if len(serialized) > self.threshold:
pickled = pickle.dumps(value)
compressed = zlib.compress(pickled, level=6)

if len(compressed) < len(serialized):
await self.redis.redis.set(
self.redis._key(key),
compressed,
ex=ttl_seconds
)
return

await self.redis.redis.set(
self.redis._key(key),
serialized,

```

```
ex = ttl_seconds
)
```

```
async def get_compression_stats(self, sample_keys: List[str]) ->
Dict:
```

```
original_size = 0
compressed_size = 0
```

```
for key in sample_keys:
    raw = await self.redis.redis.get(self.redis._key(key))
```

```
if raw is None:
    continue
```

```
compressed_size += len(raw)
```

```
if raw[:2] == b'\x78\x9c':
    decompressed = zlib.decompress(raw)
    original_size += len(decompressed)
else:
    original_size += len(raw)
```

```
ratio = compressed_size / original_size if original_size > 0 else 1.0
```

```
return {
    "original_size": original_size,
    "compressed_size": compressed_size,
    "ratio": ratio,
    "savings_percent": (1 - ratio) * 100
}
```

Compression is most effective for large JSON objects like block data and transaction receipts. Small values like balances should not be compressed.

SECTION 8: DISTRIBUTED CACHING PATTERNS

For high-availability deployments, use Redis Cluster or multiple Redis instances.

```

class DistributedCache:
    def __init__(self, redis_urls: List[str]):
        self.nodes = [redis.from_url(url) for url in redis_urls]
        self.node_count = len(self.nodes)

    def _get_node(self, key: str) -> redis.Redis:
        key_hash = int(hashlib.md5(key.encode()).hexdigest(), 16)
        node_index = key_hash % self.node_count
        return self.nodes[node_index]

    async def get(self, key: str) -> Optional[Any]:
        node = self._get_node(key)

        try:
            raw = await node.get(key)
            return json.loads(raw) if raw else None
        except Exception as e:
            print(f'Cache get error: {e}')
            return None

    async def set(
        self,
        key: str,
        value: Any,
        ttl_seconds: int = None
    ):
        node = self._get_node(key)

        try:
            await node.set(key, json.dumps(value), ex = ttl_seconds)
        except Exception as e:
            print(f'Cache set error: {e}')

    async def get_with_fallback(self, key: str) -> Optional[Any]:
        primary_node = self._get_node(key)

        try:
            raw = await primary_node.get(key)
            if raw:

```

```
return json.loads(raw)
except Exception:
    pass
```

```
for node in self.nodes:
    if node == primary_node:
        continue
```

```
try:
    raw = await node.get(key)
    if raw:
        return json.loads(raw)
except Exception:
    continue
```

```
return None
```

```
async def replicate(self, key: str, value: Any, ttl_seconds: int =
None):
    serialized = json.dumps(value)
    tasks = []
```

```
for node in self.nodes:
    tasks.append(node.set(key, serialized, ex=ttl_seconds))
```

```
await asyncio.gather(*tasks, return_exceptions=True)
```

```
async def health_check(self) -> Dict[str, bool]:
    results = {}
```

```
for i, node in enumerate(self.nodes):
    try:
        await node.ping()
        results[f'node_{i}'] = True
    except Exception:
        results[f'node_{i}'] = False
```

```
return results
```

```
async def close_all(self):
```

```
for node in self.nodes:
    await node.close()
```

Distributed caching uses consistent hashing to route keys to specific nodes. Fallback logic handles node failures gracefully.

SECTION 9: CACHING BEST PRACTICES

Here are essential practices for blockchain caching.

Never cache unconfirmed data with long TTLs. Data from recent blocks may be invalidated by chain reorganizations.

Use different TTLs for different data types. Balances need short TTLs. Historical blocks can be cached forever. Metadata can have day-long TTLs.

Implement cache stampede protection. When many requests arrive for the same uncached key, only one should fetch from the source.

```
class StampedeProtectedCache:
    def __init__(self, cache: TieredCache):
        self.cache = cache
        self.pending_fetches: Dict[str, asyncio.Future] = {}
        self.lock = asyncio.Lock()
```

```
    async def get_or_fetch(
        self,
        key: str,
        fetch_func: Callable,
        ttl_seconds: int = None
    ) -> Any:
```

```
        cached = self.cache.memory.get(key)
        if cached is not None:
            return cached
```

```
        async with self.lock:
            if key in self.pending_fetches:
```



```

return await self.pending_fetches[key]

future = asyncio.get_event_loop().create_future()
self.pending_fetches[key] = future

try:
    value = await fetch_func()

    self.cache.memory.set_pending(key, value, ttl_seconds or 30)
    await self.cache.redis.set(key, value, ttl_seconds=ttl_seconds)

    future.set_result(value)
    return value

except Exception as e:
    future.set_exception(e)
    raise

finally:
    async with self.lock:
        self.pending_fetches.pop(key, None)

```

Monitor cache hit rates. If hit rates are low, you may be caching the wrong data or using TTLs that are too short.

```

class CacheMonitor:
    def __init__(self, tiered_cache: TieredCache):
        self.cache = tiered_cache
        self.metrics_history: List[Dict] = []

    async def record_metrics(self):
        stats = self.cache.get_stats()

        self.metrics_history.append({
            "timestamp": time.time(),
            **stats
        })

    if len(self.metrics_history) > 1000:
        self.metrics_history = self.metrics_history[-1000:]

```

```

def get_hourly_summary(self) -> Dict:
    now = time.time()
    hour_ago = now - 3600

    recent = [
        m for m in self.metrics_history
        if m["timestamp"] > hour_ago
    ]

    if not recent:
        return {}

    total_requests = sum(m["total_requests"] for m in recent)
    memory_hits = sum(m["memory_hits"] for m in recent)
    redis_hits = sum(m["redis_hits"] for m in recent)
    db_hits = sum(m["db_hits"] for m in recent)
    rpc_calls = sum(m["rpc_calls"] for m in recent)

    return {
        "period": "1 hour",
        "total_requests": total_requests,
        "memory_hit_rate": memory_hits / total_requests if total_requests >
0 else 0,
        "redis_hit_rate": redis_hits / total_requests if total_requests > 0 else
0,
        "db_hit_rate": db_hits / total_requests if total_requests > 0 else 0,
        "rpc_rate": rpc_calls / total_requests if total_requests > 0 else 0,
        "estimated_rpc_savings": (total_requests - rpc_calls) if total_requests
> 0 else 0
    }

```

SECTION 10: PUTTING IT ALL TOGETHER

Here is a complete caching setup for a blockchain application.

```

async def create_caching_layer(config: Dict) -> TieredCache:
    memory_cache =
BlockAwareCache(confirmations = config.get("confirmations", 12))

```

```
redis_cache = BlockchainRedisCache(config["redis_url"])
```

```
rpc_client = ResilientRPCClient(config["rpc_providers"])
```

```
from indexer.database import IndexerDatabase
```

```
database = IndexerDatabase(config["database_url"])
```

```
await database.connect()
```

```
tiered_cache = TieredCache(  
    memory_cache=memory_cache,  
    redis_cache=redis_cache,  
    rpc_client=rpc_client,  
    database=database  
)
```

```
invalidator = TransferInvalidator(memory_cache, redis_cache)
```

```
warmer = CacheWarmer(tiered_cache, database, rpc_client)
```

```
monitor = CacheMonitor(tiered_cache)
```

```
asyncio.create_task(warmer.run_warming_schedule())
```

```
async def record_metrics_loop():
```

```
    while True:
```

```
        await monitor.record_metrics()
```

```
        await asyncio.sleep(60)
```

```
asyncio.create_task(record_metrics_loop())
```

```
return tiered_cache
```

```
async def main():
```

```
    config = {
```

```
        "redis_url": "redis://localhost:6379",
```

```
        "database_url": "postgresql://user:pass@localhost/blockchain",
```

```
        "rpc_providers": [
```

```
            {"name": "primary", "url": "https://eth-mainnet.example.com",
```

```
"priority": 0}
],
"confirmations": 12
}
```

```
cache = await create_caching_layer(config)
```

```
balance = await cache.get_balance(
    "0x742d35Cc6634C0532925a3b844Bc9e7595f3bC28"
)
print(f'ETH Balance: {balance}')
```

```
owner = await cache.get_nft_owner(
    "0xBC4CA0EdA7647A8aB7C2061c2E118A18a936f13D",
    "1234"
)
print(f'NFT Owner: {owner}')
```

```
stats = cache.get_stats()
print(f'Cache stats: {stats}')
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

This chapter covered comprehensive caching strategies for blockchain applications. You learned in-memory LRU caching for hot data, Redis caching for shared state, block-aware caching that handles confirmations, cache invalidation based on blockchain events, tiered caching that checks multiple layers, cache warming to avoid cold starts, compression for large objects, and distributed caching for high availability. Proper caching transforms slow, expensive RPC calls into fast, cheap cache hits.

CHAPTER 4: TRANSACTION MANAGEMENT

Sending transactions reliably is harder than reading blockchain data. Nonces must be sequential without gaps. Gas prices affect confirmation time and cost. Transactions can get stuck in the mempool. Multiple services sending from the same wallet create race conditions. This chapter teaches production-grade transaction management.

SECTION 1: THE TRANSACTION LIFECYCLE

A transaction goes through distinct phases.

Construction is where you build the transaction with recipient, value, data, gas limit, and gas price.

Signing is where you sign the transaction with the sender's private key.

Broadcasting is where you send the raw signed transaction to an RPC node.

Pending is where the transaction sits in the mempool waiting for inclusion.

Included is where a miner includes the transaction in a block.

Confirmed is where enough blocks have been mined on top. Usually 12 confirmations.

Things can go wrong at every phase. Construction can fail with invalid data. Signing can fail with wrong keys. Broadcasting can fail with network errors. Pending can fail with gas too low. Inclusion can fail with nonce issues. Confirmation can fail with chain reorganization.

Production systems must handle all failure modes.

SECTION 2: NONCE MANAGEMENT

Every transaction has a nonce, a sequential number starting from zero. Nonces must be used in order without gaps. If you send nonce 5 before nonce 4 confirms, nonce 5 will be stuck waiting for nonce 4.

`import asyncio`

```
from typing import Dict, Optional, List
from dataclasses import dataclass, field
from collections import deque
import time
```

```
@dataclass
class NonceSlot:
    nonce: int
    transaction_hash: Optional[str] = None
    status: str = "pending"
    created_at: float = field(default_factory = time.time)
    attempts: int = 0
```

```
class NonceManager:
    def __init__(self, rpc_client, address: str):
        self.rpc = rpc_client
        self.address = address.lower()
        self.lock = asyncio.Lock()
        self.local_nonce: Optional[int] = None
        self.pending_nonces: Dict[int, NonceSlot] = {}
        self.confirmed_nonce: Optional[int] = None
```

```
    async def initialize(self):
        async with self.lock:
            pending_count = await self.rpc.call(
                "eth_getTransactionCount",
                [self.address, "pending"]
            )
            confirmed_count = await self.rpc.call(
                "eth_getTransactionCount",
                [self.address, "latest"]
            )
```

```
        self.local_nonce = int(pending_count, 16)
        self.confirmed_nonce = int(confirmed_count, 16)
```

```
        print(f"Initialized nonce manager:
        confirmed = {self.confirmed_nonce}, pending = {self.local_nonce}")
```

```
    async def get_next_nonce(self) -> int:
```

```

async with self.lock:
if self.local_nonce is None:
await self.initialize()

nonce = self.local_nonce
self.local_nonce += 1

self.pending_nonces[nonce] = NonceSlot(nonce=nonce)

return nonce

async def release_nonce(self, nonce: int):
async with self.lock:
if nonce in self.pending_nonces:
del self.pending_nonces[nonce]

if nonce + 1 == self.local_nonce:
self.local_nonce = nonce

async def mark_submitted(self, nonce: int, tx_hash: str):
async with self.lock:
if nonce in self.pending_nonces:
self.pending_nonces[nonce].transaction_hash = tx_hash
self.pending_nonces[nonce].status = "submitted"

async def mark_confirmed(self, nonce: int):
async with self.lock:
if nonce in self.pending_nonces:
del self.pending_nonces[nonce]

if self.confirmed_nonce is None or nonce > =
self.confirmed_nonce:
self.confirmed_nonce = nonce + 1

async def sync_with_chain(self):
async with self.lock:
pending_count = await self.rpc.call(
"eth_getTransactionCount",
[self.address, "pending"]
)

```

```

confirmed_count = await self.rpc.call(
    "eth_getTransactionCount",
    [self.address, "latest"]
)

chain_pending = int(pending_count, 16)
chain_confirmed = int(confirmed_count, 16)

for nonce in list(self.pending_nonces.keys()):
    if nonce < chain_confirmed:
        del self.pending_nonces[nonce]

    if self.local_nonce is not None and chain_pending >
self.local_nonce:
        print(f'Chain advanced nonce: {self.local_nonce} ->
{chain_pending}')
        self.local_nonce = chain_pending

self.confirmed_nonce = chain_confirmed

def get_pending_count(self) -> int:
    return len(self.pending_nonces)

def get_gap_nonces(self) -> List[int]:
    if not self.pending_nonces:
        return []

    min_nonce = min(self.pending_nonces.keys())
    max_nonce = max(self.pending_nonces.keys())

    expected = set(range(min_nonce, max_nonce + 1))
    actual = set(self.pending_nonces.keys())

    return sorted(expected - actual)

```

The nonce manager tracks both local and chain nonces. It handles gaps, synchronization, and concurrent access safely.

SECTION 3: GAS PRICE STRATEGIES

Gas price determines how quickly transactions confirm and how much they cost. Too low and transactions get stuck. Too high and you waste money.

```
from enum import Enum
from typing import Tuple
import statistics
```

```
class GasStrategy(Enum):
    SLOW = "slow"
    STANDARD = "standard"
    FAST = "fast"
    INSTANT = "instant"
```

```
class GasPriceOracle:
    def __init__(self, rpc_client):
        self.rpc = rpc_client
        self.price_history: deque = deque(maxlen=100)
        self.last_update: float = 0
        self.update_interval: float = 10.0
```

```
    async def get_gas_price(self, strategy: GasStrategy =
GasStrategy.STANDARD) -> int:
        await self._update_if_needed()
```

```
        base_price = await self._get_current_base_price()
```

```
        multipliers = {
            GasStrategy.SLOW: 0.9,
            GasStrategy.STANDARD: 1.0,
            GasStrategy.FAST: 1.25,
            GasStrategy.INSTANT: 1.5
        }
```

```
        price = int(base_price * multipliers[strategy])
```

```
        min_price = 1_000_000_000
        return max(price, min_price)
```

```
async def _get_current_base_price(self) -> int:
    result = await self.rpc.call("eth_gasPrice")
    return int(result, 16)
```

```
async def _update_if_needed(self):
    now = time.time()
```

```
    if now - self.last_update < self.update_interval:
        return
```

```
    price = await self._get_current_base_price()
    self.price_history.append({
        "timestamp": now,
        "price": price
    })
```

```
    self.last_update = now
```

```
def get_price_trend(self) -> str:
    if len(self.price_history) < 10:
        return "unknown"
```

```
    recent = list(self.price_history)[-10:]
    older = list(self.price_history)[-20:-10] if len(self.price_history)
    >= 20 else recent
```

```
    recent_avg = statistics.mean(p["price"] for p in recent)
    older_avg = statistics.mean(p["price"] for p in older)
```

```
    if recent_avg > older_avg * 1.1:
        return "increasing"
    elif recent_avg < older_avg * 0.9:
        return "decreasing"
    else:
        return "stable"
```

```
class EIP1559GasOracle(GasPriceOracle):
    async def get_gas_params(
        self,
```

```
strategy: GasStrategy = GasStrategy.STANDARD
) -> Dict[str, int]:
```

```
latest_block = await self.rpc.call("eth_getBlockByNumber",
["latest", False])
```

```
base_fee = int(latest_block["baseFeePerGas"], 16)
```

```
priority_fees = {
    GasStrategy.SLOW: 1_000_000_000,
    GasStrategy.STANDARD: 1_500_000_000,
    GasStrategy.FAST: 2_500_000_000,
    GasStrategy.INSTANT: 5_000_000_000
}
```

```
priority_fee = priority_fees[strategy]
```

```
buffer_multipliers = {
    GasStrategy.SLOW: 1.1,
    GasStrategy.STANDARD: 1.25,
    GasStrategy.FAST: 1.5,
    GasStrategy.INSTANT: 2.0
}
```

```
max_fee = int(base_fee * buffer_multipliers[strategy]) +
priority_fee
```

```
return {
    "maxFeePerGas": max_fee,
    "maxPriorityFeePerGas": priority_fee
}
```

```
async def estimate_cost(
    self,
    gas_limit: int,
    strategy: GasStrategy = GasStrategy.STANDARD
) -> Dict[str, int]:
```

```
params = await self.get_gas_params(strategy)
```

```
max_cost = gas_limit * params["maxFeePerGas"]
```

```
latest_block = await self.rpc.call("eth_getBlockByNumber",  
["latest", False])
```

```
base_fee = int(latest_block["baseFeePerGas"], 16)
```

```
likely_cost = gas_limit * (base_fee +  
params["maxPriorityFeePerGas"])
```

```
return {  
    "max_cost_wei": max_cost,  
    "likely_cost_wei": likely_cost,  
    "max_cost_gwei": max_cost / 1e9,  
    "likely_cost_gwei": likely_cost / 1e9  
}
```

EIP-1559 transactions use dynamic base fees plus priority fees. The oracle calculates appropriate values for different speed requirements.

SECTION 4: TRANSACTION BUILDER

Construct transactions with proper gas estimation and data encoding.

```
from eth_abi import encode
```

```
from eth_utils import keccak, function_signature_to_4byte_selector
```

```
@dataclass
```

```
class TransactionParams:
```

```
    to: str
```

```
    value: int = 0
```

```
    data: str = "0x"
```

```
    gas_limit: Optional[int] = None
```

```
    gas_price: Optional[int] = None
```

```
    max_fee_per_gas: Optional[int] = None
```

```
    max_priority_fee_per_gas: Optional[int] = None
```

```
    nonce: Optional[int] = None
```

```
class TransactionBuilder:
```

```

def __init__(
    self,
    rpc_client,
    nonce_manager: NonceManager,
    gas_oracle: EIP1559GasOracle
):
    self.rpc = rpc_client
    self.nonce_manager = nonce_manager
    self.gas_oracle = gas_oracle
    self.chain_id: Optional[int] = None

    async def initialize(self):
        result = await self.rpc.call("eth_chainId")
        self.chain_id = int(result, 16)

    async def build_transaction(
        self,
        params: TransactionParams,
        strategy: GasStrategy = GasStrategy.STANDARD
    ) -> Dict:

        if self.chain_id is None:
            await self.initialize()

        tx = {
            "to": params.to,
            "value": hex(params.value),
            "data": params.data,
            "chainId": hex(self.chain_id)
        }

        if params.nonce is not None:
            tx["nonce"] = hex(params.nonce)
        else:
            nonce = await self.nonce_manager.get_next_nonce()
            tx["nonce"] = hex(nonce)

        if params.gas_limit is not None:
            tx["gas"] = hex(params.gas_limit)
        else:

```

```
gas_estimate = await self._estimate_gas(params)
gas_with_buffer = int(gas_estimate * 1.2)
tx["gas"] = hex(gas_with_buffer)
```

```
if params.max_fee_per_gas is not None:
    tx["maxFeePerGas"] = hex(params.max_fee_per_gas)
    tx["maxPriorityFeePerGas"] = hex(
        params.max_priority_fee_per_gas or 1_500_000_000
    )
```

```
tx["type"] = "0x2"
```

```
elif params.gas_price is not None:
    tx["gasPrice"] = hex(params.gas_price)
```

```
else:
```

```
    gas_params = await self.gas_oracle.get_gas_params(strategy)
```

```
    tx["maxFeePerGas"] = hex(gas_params["maxFeePerGas"])
```

```
    tx["maxPriorityFeePerGas"] =
```

```
    hex(gas_params["maxPriorityFeePerGas"])
```

```
    tx["type"] = "0x2"
```

```
return tx
```

```
async def _estimate_gas(self, params: TransactionParams) -> int:
```

```
    estimate_params = {
```

```
        "to": params.to,
```

```
        "data": params.data
```

```
    }
```

```
if params.value > 0:
```

```
    estimate_params["value"] = hex(params.value)
```

```
result = await self.rpc.call("eth_estimateGas", [estimate_params])
```

```
return int(result, 16)
```

```
def encode_function_call(
```

```
    self,
```

```
    function_signature: str,
```

```
    args: List[Any]
```

```
) -> str:
```

```
selector = "0x" + keccak(text=function_signature)[:4].hex()
```

```
if "(" not in function_signature or ")" in function_signature:  
    return selector
```

```
types_str = function_signature.split("(")[1].rstrip("  
types = [t.strip() for t in types_str.split(",") if t.strip()]
```

```
encoded_args = encode(types, args)
```

```
return selector + encoded_args.hex()
```

```
async def build_erc20_transfer(  
    self,  
    token_address: str,  
    recipient: str,  
    amount: int,  
    strategy: GasStrategy = GasStrategy.STANDARD  
) -> Dict:
```

```
    data = self.encode_function_call(  
        "transfer(address,uint256)",  
        [recipient, amount]  
    )
```

```
    params = TransactionParams(  
        to = token_address,  
        data = data  
    )
```

```
    return await self.build_transaction(params, strategy)
```

```
async def build_erc721_transfer(  
    self,  
    token_address: str,  
    from_address: str,  
    to_address: str,  
    token_id: int,  
    strategy: GasStrategy = GasStrategy.STANDARD  
) -> Dict:
```

```

data = self.encode_function_call(
    "safeTransferFrom(address,address,uint256)",
    [from_address, to_address, token_id]
)

params = TransactionParams(
    to = token_address,
    data = data
)

return await self.build_transaction(params, strategy)

async def build_contract_call(
    self,
    contract_address: str,
    function_signature: str,
    args: List[Any],
    value: int = 0,
    strategy: GasStrategy = GasStrategy.STANDARD
) -> Dict:

    data = self.encode_function_call(function_signature, args)

    params = TransactionParams(
        to = contract_address,
        value = value,
        data = data
    )

    return await self.build_transaction(params, strategy)

```

The transaction builder handles gas estimation, nonce assignment, and function encoding. It produces ready-to-sign transaction objects.

SECTION 5: TRANSACTION SIGNING AND SENDING

Sign transactions securely and broadcast them to the network.


```
from eth_account import Account
from eth_account.signers.local import LocalAccount
import os
```

```
class TransactionSigner:
    def __init__(self, private_key: str = None):
        if private_key:
            self.account: LocalAccount = Account.from_key(private_key)
        else:
            key = os.environ.get("PRIVATE_KEY")
            if not key:
                raise ValueError("No private key provided")
            self.account = Account.from_key(key)

        self.address = self.account.address

    def sign_transaction(self, tx: Dict) -> str:
        signed = self.account.sign_transaction(tx)
        return signed.rawTransaction.hex()
```

```
    def sign_message(self, message: bytes) -> str:
        signed = self.account.sign_message(message)
        return signed.signature.hex()
```

```
class TransactionSender:
    def __init__(
        self,
        rpc_client,
        signer: TransactionSigner,
        nonce_manager: NonceManager
    ):
        self.rpc = rpc_client
        self.signer = signer
        self.nonce_manager = nonce_manager
        self.pending_transactions: Dict[str, Dict] = {}

    async def send_transaction(self, tx: Dict) -> str:
        nonce = int(tx["nonce"], 16)
```

```

try:
    raw_tx = self.signer.sign_transaction(tx)

    tx_hash = await self.rpc.call(
        "eth_sendRawTransaction",
        ["0x" + raw_tx if not raw_tx.startswith("0x") else raw_tx]
    )

    await self.nonce_manager.mark_submitted(nonce, tx_hash)

    self.pending_transactions[tx_hash] = {
        "tx": tx,
        "nonce": nonce,
        "submitted_at": time.time(),
        "attempts": 1
    }

    return tx_hash

except Exception as e:
    error_message = str(e).lower()

    if "nonce too low" in error_message:
        await self.nonce_manager.sync_with_chain()
        raise NonceError("Nonce too low, sync required")

    elif "replacement transaction underpriced" in error_message:
        raise GasPriceError("Gas price too low for replacement")

    elif "insufficient funds" in error_message:
        raise InsufficientFundsError("Not enough ETH for transaction")

    else:
        await self.nonce_manager.release_nonce(nonce)
        raise

    async def wait_for_confirmation(
        self,
        tx_hash: str,
        confirmations: int = 1,

```

```
timeout: int = 300
```

```
) -> Dict:
```

```
start_time = time.time()
```

```
while time.time() - start_time < timeout:
```

```
    receipt = await self.rpc.call(  
        "eth_getTransactionReceipt",  
        [tx_hash]  
    )
```

```
    if receipt is not None:
```

```
        tx_block = int(receipt["blockNumber"], 16)
```

```
        current_block = await self.rpc.call("eth_blockNumber")
```

```
        current_block = int(current_block, 16)
```

```
    if current_block - tx_block >= confirmations:
```

```
        if tx_hash in self.pending_transactions:
```

```
            nonce = self.pending_transactions[tx_hash]["nonce"]
```

```
            await self.nonce_manager.mark_confirmed(nonce)
```

```
            del self.pending_transactions[tx_hash]
```

```
        return {
```

```
            "status": "confirmed",
```

```
            "receipt": receipt,
```

```
            "confirmations": current_block - tx_block
```

```
        }
```

```
    await asyncio.sleep(2)
```

```
    return {
```

```
        "status": "timeout",
```

```
        "tx_hash": tx_hash
```

```
    }
```

```
    async def send_and_wait(  
        self,
```

```
        tx: Dict,  
        confirmations: int = 1,
```

```
        timeout: int = 300
```

) -> Dict:

```
tx_hash = await self.send_transaction(tx)
```

```
result = await self.wait_for_confirmation(tx_hash, confirmations,  
timeout)
```

```
return {  
    "tx_hash": tx_hash,  
    **result  
}
```

```
class NonceError(Exception):  
    pass
```

```
class GasPriceError(Exception):  
    pass
```

```
class InsufficientFundsError(Exception):  
    pass
```

The sender handles signing, broadcasting, and confirmation waiting. Error handling distinguishes between different failure types.

SECTION 6: TRANSACTION QUEUE

For high-volume applications, queue transactions and process them sequentially.

```
from enum import Enum  
from typing import Callable, Awaitable
```

```
class TransactionStatus(Enum):  
    QUEUED = "queued"  
    BUILDING = "building"  
    PENDING = "pending"  
    CONFIRMED = "confirmed"
```

FAILED = "failed"

```
@dataclass
class QueuedTransaction:
    id: str
    params: TransactionParams
    strategy: GasStrategy
    status: TransactionStatus = TransactionStatus.QUEUED
    tx_hash: Optional[str] = None
    receipt: Optional[Dict] = None
    error: Optional[str] = None
    created_at: float = field(default_factory=time.time)
    attempts: int = 0
    max_attempts: int = 3
    callback: Optional[Callable] = None
```

```
class TransactionQueue:
    def __init__(
        self,
        builder: TransactionBuilder,
        sender: TransactionSender,
        max_pending: int = 5
    ):
        self.builder = builder
        self.sender = sender
        self.max_pending = max_pending
        self.queue: deque[QueuedTransaction] = deque()
        self.processing: Dict[str, QueuedTransaction] = {}
        self.completed: Dict[str, QueuedTransaction] = {}
        self.running = False
        self.lock = asyncio.Lock()
```

```
    async def add(
        self,
        params: TransactionParams,
        strategy: GasStrategy = GasStrategy.STANDARD,
        callback: Callable = None
    ) -> str:
```

```
        tx_id = f'tx_{int(time.time() * 1000)}_{len(self.queue)}"
```

```
queued_tx = QueuedTransaction(  
    id = tx_id,  
    params = params,  
    strategy = strategy,  
    callback = callback  
)
```

```
async with self.lock:  
    self.queue.append(queued_tx)
```

```
return tx_id
```

```
async def start(self):  
    self.running = True
```

```
while self.running:  
    await self._process_queue()  
    await asyncio.sleep(0.5)
```

```
async def stop(self):  
    self.running = False
```

```
async def _process_queue(self):  
    async with self.lock:  
        pending_count = len([  
            tx for tx in self.processing.values()  
            if tx.status == TransactionStatus.PENDING  
        ])
```

```
while self.queue and pending_count < self.max_pending:  
    queued_tx = self.queue.popleft()  
    self.processing[queued_tx.id] = queued_tx  
    asyncio.create_task(self._process_transaction(queued_tx))  
    pending_count += 1
```

```
async def _process_transaction(self, queued_tx: QueuedTransaction):  
    try:  
        queued_tx.status = TransactionStatus.BUILDING  
        queued_tx.attempts += 1
```

```
tx = await self.builder.build_transaction(  
    queued_tx.params,  
    queued_tx.strategy  
)
```

```
queued_tx.status = TransactionStatus.PENDING
```

```
result = await self.sender.send_and_wait(tx, confirmations = 1,  
    timeout = 300)
```

```
if result["status"] == "confirmed":  
    queued_tx.status = TransactionStatus.CONFIRMED  
    queued_tx.tx_hash = result["tx_hash"]  
    queued_tx.receipt = result["receipt"]  
else:  
    raise Exception(f"Transaction not confirmed: {result}")
```

```
except NonceError as e:  
    if queued_tx.attempts < queued_tx.max_attempts:  
        async with self.lock:  
            self.queue.appendleft(queued_tx)  
            del self.processing[queued_tx.id]  
        return  
    else:  
        queued_tx.status = TransactionStatus.FAILED  
        queued_tx.error = str(e)
```

```
except Exception as e:  
    if queued_tx.attempts < queued_tx.max_attempts:  
        async with self.lock:  
            self.queue.appendleft(queued_tx)  
            del self.processing[queued_tx.id]  
        return  
    else:  
        queued_tx.status = TransactionStatus.FAILED  
        queued_tx.error = str(e)
```

```
async with self.lock:  
    del self.processing[queued_tx.id]
```

```
self.completed[queued_tx.id] = queued_tx
```

```
if queued_tx.callback:
```

```
try:
```

```
if asyncio.iscoroutinefunction(queued_tx.callback):
```

```
await queued_tx.callback(queued_tx)
```

```
else:
```

```
queued_tx.callback(queued_tx)
```

```
except Exception as e:
```

```
print(f'Callback error: {e}')
```

```
def get_status(self, tx_id: str) -> Optional[QueuedTransaction]:
```

```
if tx_id in self.processing:
```

```
return self.processing[tx_id]
```

```
if tx_id in self.completed:
```

```
return self.completed[tx_id]
```

```
for tx in self.queue:
```

```
if tx.id == tx_id:
```

```
return tx
```

```
return None
```

```
def get_queue_stats(self) -> Dict:
```

```
return {
```

```
"queued": len(self.queue),
```

```
"processing": len(self.processing),
```

```
"completed": len(self.completed),
```

```
"failed": len([
```

```
tx for tx in self.completed.values()
```

```
if tx.status == TransactionStatus.FAILED
```

```
])
```

```
}
```

The queue manages transaction flow with configurable concurrency. Retries handle transient failures automatically.

SECTION 7: STUCK TRANSACTION RECOVERY

Transactions can get stuck when gas prices rise or network congestion increases. Recovery requires replacing or canceling stuck transactions.

```
class StuckTransactionRecovery:
    def __init__(
        self,
        rpc_client,
        sender: TransactionSender,
        gas_oracle: EIP1559GasOracle
    ):
        self.rpc = rpc_client
        self.sender = sender
        self.gas_oracle = gas_oracle

    async def find_stuck_transactions(
        self,
        address: str,
        age_threshold: int = 300
    ) -> List[Dict]:

        pending_nonce = await self.rpc.call(
            "eth_getTransactionCount",
            [address, "pending"]
        )
        confirmed_nonce = await self.rpc.call(
            "eth_getTransactionCount",
            [address, "latest"]
        )

        pending_nonce = int(pending_nonce, 16)
        confirmed_nonce = int(confirmed_nonce, 16)

        stuck = []

        for tx_hash, tx_data in self.sender.pending_transactions.items():
            age = time.time() - tx_data["submitted_at"]

            if age > age_threshold:
                stuck.append({
```

```
"tx_hash": tx_hash,  
"nonce": tx_data["nonce"],  
"age_seconds": age,  
"tx": tx_data["tx"]  
})
```

return stuck

```
async def speed_up_transaction(  
self,  
original_tx: Dict,  
gas_multiplier: float = 1.5  
) -> str:
```

```
nonce = original_tx["nonce"]
```

```
gas_params = await  
self.gas_oracle.get_gas_params(GasStrategy.INSTANT)
```

```
if "maxFeePerGas" in original_tx:  
original_max_fee = int(original_tx["maxFeePerGas"], 16)  
original_priority = int(original_tx["maxPriorityFeePerGas"], 16)
```

```
new_max_fee = max(  
int(original_max_fee * gas_multiplier),  
gas_params["maxFeePerGas"]  
)  
new_priority = max(  
int(original_priority * gas_multiplier),  
gas_params["maxPriorityFeePerGas"]  
)
```

```
new_tx = {  
**original_tx,  
"maxFeePerGas": hex(new_max_fee),  
"maxPriorityFeePerGas": hex(new_priority)  
}  
else:  
original_gas_price = int(original_tx["gasPrice"], 16)  
new_gas_price = int(original_gas_price * gas_multiplier)
```

```

new_tx = {
**original_tx,
"gasPrice": hex(new_gas_price)
}

```

```

raw_tx = self.sender.signer.sign_transaction(new_tx)
new_hash = await self.rpc.call(
"eth_sendRawTransaction",
["0x" + raw_tx if not raw_tx.startswith("0x") else raw_tx]
)

```

```

return new_hash

```

```

async def cancel_transaction(
self,
address: str,
nonce: int
) -> str:

```

```

gas_params = await
self.gas_oracle.get_gas_params(GasStrategy.INSTANT)

```

```

cancel_tx = {
"to": address,
"value": "0x0",
"data": "0x",
"nonce": hex(nonce),
"gas": hex(21000),
"maxFeePerGas": hex(gas_params["maxFeePerGas"]),
"maxPriorityFeePerGas": hex(gas_params["maxPriorityFeePerGas"]),
"type": "0x2",
"chainId": hex(self.sender.signer.account.chain_id or 1)
}

```

```

raw_tx = self.sender.signer.sign_transaction(cancel_tx)
cancel_hash = await self.rpc.call(
"eth_sendRawTransaction",
["0x" + raw_tx if not raw_tx.startswith("0x") else raw_tx]
)

```

```
return cancel_hash
```

```
async def recover_all_stuck(  
    self,  
    address: str,  
    strategy: str = "speed_up"  
    ) -> List[Dict]:
```

```
    stuck = await self.find_stuck_transactions(address)  
    results = []
```

```
    for stuck_tx in stuck:
```

```
        try:
```

```
            if strategy == "speed_up":
```

```
                new_hash = await self.speed_up_transaction(stuck_tx["tx"])
```

```
                results.append({
```

```
                    "original_hash": stuck_tx["tx_hash"],
```

```
                    "new_hash": new_hash,
```

```
                    "action": "speed_up"
```

```
                })
```

```
            elif strategy == "cancel":
```

```
                cancel_hash = await self.cancel_transaction(  
                    address,
```

```
                    stuck_tx["nonce"]  
                )
```

```
                results.append({
```

```
                    "original_hash": stuck_tx["tx_hash"],
```

```
                    "cancel_hash": cancel_hash,
```

```
                    "action": "canceled"
```

```
                })
```

```
        except Exception as e:
```

```
            results.append({
```

```
                "original_hash": stuck_tx["tx_hash"],
```

```
                "error": str(e),
```

```
                "action": "failed"
```

```
            })
```

```
    return results
```

Stuck transaction recovery can speed up transactions with higher gas or cancel them by sending a zero-value transaction with the same nonce.

SECTION 8: MULTI-WALLET MANAGEMENT

Large applications often need multiple wallets for different purposes.

```
@dataclass
class WalletConfig:
    name: str
    address: str
    private_key: str
    purpose: str
    daily_limit_wei: int = 0
    min_balance_wei: int = 0

class MultiWalletManager:
    def __init__(self, rpc_client, builder: TransactionBuilder):
        self.rpc = rpc_client
        self.builder = builder
        self.wallets: Dict[str, WalletConfig] = {}
        self.nonce_managers: Dict[str, NonceManager] = {}
        self.signers: Dict[str, TransactionSigner] = {}
        self.senders: Dict[str, TransactionSender] = {}
        self.daily_spent: Dict[str, int] = {}
        self.last_reset: Dict[str, float] = {}

    async def add_wallet(self, config: WalletConfig):
        name = config.name

        self.wallets[name] = config

        nonce_manager = NonceManager(self.rpc, config.address)
        await nonce_manager.initialize()
        self.nonce_managers[name] = nonce_manager
```

```
signer = TransactionSigner(config.private_key)
self.signers[name] = signer
```

```
sender = TransactionSender(self.rpc, signer, nonce_manager)
self.senders[name] = sender
```

```
self.daily_spent[name] = 0
self.last_reset[name] = time.time()
```

```
async def get_wallet_status(self, name: str) -> Dict:
if name not in self.wallets:
return {"error": "Wallet not found"}
```

```
config = self.wallets[name]
```

```
balance = await self.rpc.call("eth_getBalance", [config.address,
"latest"])
balance = int(balance, 16)
```

```
nonce_manager = self.nonce_managers[name]
pending_count = nonce_manager.get_pending_count()
```

```
now = time.time()
if now - self.last_reset[name] > 86400:
self.daily_spent[name] = 0
self.last_reset[name] = now
```

```
return {
"name": name,
"address": config.address,
"purpose": config.purpose,
"balance_wei": balance,
"balance_eth": balance / 1e18,
"pending_transactions": pending_count,
"daily_spent_wei": self.daily_spent[name],
"daily_limit_wei": config.daily_limit_wei,
"daily_remaining_wei": config.daily_limit_wei -
self.daily_spent[name],
"min_balance_ok": balance >= config.min_balance_wei
}
```

```

async def select_wallet(
    self,
    purpose: str,
    value_wei: int = 0
) -> Optional[str]:

    candidates = []

    for name, config in self.wallets.items():
        if config.purpose != purpose:
            continue

        status = await self.get_wallet_status(name)

        if not status.get("min_balance_ok"):
            continue

        if config.daily_limit_wei > 0:
            if self.daily_spent[name] + value_wei > config.daily_limit_wei:
                continue

        candidates.append((name, status["pending_transactions"]))

    if not candidates:
        return None

    candidates.sort(key = lambda x: x[1])
    return candidates[0][0]

async def send_from_wallet(
    self,
    wallet_name: str,
    params: TransactionParams,
    strategy: GasStrategy = GasStrategy.STANDARD
) -> str:

    if wallet_name not in self.wallets:
        raise ValueError(f'Unknown wallet: {wallet_name}')

```

```
nonce_manager = self.nonce_managers[wallet_name]
original_nonce_manager = self.builder.nonce_manager
self.builder.nonce_manager = nonce_manager
```

```
try:
```

```
tx = await self.builder.build_transaction(params, strategy)
```

```
sender = self.senders[wallet_name]
```

```
tx_hash = await sender.send_transaction(tx)
```

```
cost_estimate = int(tx["gas"], 16) * int(tx.get("maxFeePerGas",
tx.get("gasPrice", "0x0")), 16)
```

```
self.daily_spent[wallet_name] += params.value + cost_estimate
```

```
return tx_hash
```

```
finally:
```

```
self.builder.nonce_manager = original_nonce_manager
```

```
async def refill_wallet(
```

```
self,
```

```
target_wallet: str,
```

```
source_wallet: str,
```

```
amount_wei: int
```

```
) -> str:
```

```
if target_wallet not in self.wallets:
```

```
raise ValueError(f"Unknown target wallet: {target_wallet}")
```

```
if source_wallet not in self.wallets:
```

```
raise ValueError(f"Unknown source wallet: {source_wallet}")
```

```
target_config = self.wallets[target_wallet]
```

```
params = TransactionParams(
```

```
to= target_config.address,
```

```
value= amount_wei
```

```
)
```

```
return await self.send_from_wallet(source_wallet, params)
```



```

async def balance_wallets(
    self,
    purpose: str,
    target_balance: int,
    source_wallet: str
) -> List[Dict]:

    results = []

    for name, config in self.wallets.items():
        if config.purpose != purpose:
            continue

        status = await self.get_wallet_status(name)
        current = status["balance_wei"]

        if current < target_balance:
            needed = target_balance - current

            try:
                tx_hash = await self.refill_wallet(name, source_wallet, needed)
                results.append({
                    "wallet": name,
                    "amount": needed,
                    "tx_hash": tx_hash,
                    "status": "success"
                })
            except Exception as e:
                results.append({
                    "wallet": name,
                    "amount": needed,
                    "error": str(e),
                    "status": "failed"
                })

    return results

```

Multi-wallet management enables wallet rotation, purpose separation, and automatic rebalancing.

SECTION 9: TRANSACTION MONITORING

Track transaction status across the lifecycle.

```
class TransactionMonitor:
    def __init__(self, rpc_client, redis_cache):
        self.rpc = rpc_client
        self.cache = redis_cache
        self.watchers: Dict[str, asyncio.Task] = {}
        self.callbacks: Dict[str, List[Callable]] = {}

    async def watch_transaction(
        self,
        tx_hash: str,
        callback: Callable = None,
        timeout: int = 600
    ):
        if callback:
            if tx_hash not in self.callbacks:
                self.callbacks[tx_hash] = []
            self.callbacks[tx_hash].append(callback)

        if tx_hash in self.watchers:
            return

        task = asyncio.create_task(
            self._monitor_transaction(tx_hash, timeout)
        )
        self.watchers[tx_hash] = task

    async def _monitor_transaction(self, tx_hash: str, timeout: int):
        start_time = time.time()
        last_status = None

        try:
            while time.time() - start_time < timeout:
                status = await self._get_transaction_status(tx_hash)
```

```

if status != last_status:
    await self._notify_callbacks(tx_hash, status)
    await self._cache_status(tx_hash, status)
    last_status = status

if status["state"] in ["confirmed", "failed"]:
    break

await asyncio.sleep(2)

if last_status["state"] == "pending":
    await self._notify_callbacks(tx_hash, {
        "state": "timeout",
        "tx_hash": tx_hash
    })

finally:
    del self.watchers[tx_hash]
    self.callbacks.pop(tx_hash, None)

async def _get_transaction_status(self, tx_hash: str) -> Dict:
    tx = await self.rpc.call("eth_getTransactionByHash", [tx_hash])

    if tx is None:
        return {
            "state": "unknown",
            "tx_hash": tx_hash
        }

    receipt = await self.rpc.call("eth_getTransactionReceipt",
    [tx_hash])

    if receipt is None:
        return {
            "state": "pending",
            "tx_hash": tx_hash,
            "nonce": int(tx["nonce"], 16),
            "gas_price": int(tx.get("gasPrice", tx.get("maxFeePerGas", "0x0")),
            16)
        }

```

```
status = int(receipt["status"], 16)
block_number = int(receipt["blockNumber"], 16)

current_block = await self.rpc.call("eth_blockNumber")
current_block = int(current_block, 16)
confirmations = current_block - block_number
```

```
if status == 1:
    return {
        "state": "confirmed",
        "tx_hash": tx_hash,
        "block_number": block_number,
        "confirmations": confirmations,
        "gas_used": int(receipt["gasUsed"], 16)
    }
else:
    return {
        "state": "failed",
        "tx_hash": tx_hash,
        "block_number": block_number,
        "revert_reason": await self._get_revert_reason(tx_hash)
    }
```

```
async def _get_revert_reason(self, tx_hash: str) -> Optional[str]:
    try:
        tx = await self.rpc.call("eth_getTransactionByHash", [tx_hash])
```

```
    if not tx:
        return None
```

```
    await self.rpc.call("eth_call", [
        {
            "to": tx["to"],
            "data": tx["input"],
            "from": tx["from"],
            "gas": tx["gas"],
            "gasPrice": tx.get("gasPrice"),
            "value": tx["value"]
        },
```

```
hex(int(tx["blockNumber"], 16))
])
```

```
return None
```

```
except Exception as e:
    error_message = str(e)
```

```
if "execution reverted" in error_message:
    if "revert " in error_message:
        return error_message.split("revert ")[1].strip("")
    return "Unknown revert reason"
```

```
return error_message
```

```
async def _notify_callbacks(self, tx_hash: str, status: Dict):
    callbacks = self.callbacks.get(tx_hash, [])
```

```
for callback in callbacks:
    try:
        if asyncio.iscoroutinefunction(callback):
            await callback(status)
        else:
            callback(status)
    except Exception as e:
        print(f"Callback error: {e}")
```

```
async def _cache_status(self, tx_hash: str, status: Dict):
    cache_key = f'tx_status:{tx_hash}'
    ttl = 3600 if status["state"] in ["confirmed", "failed"] else 60
    await self.cache.set(cache_key, status, ttl_seconds=ttl)
```

```
async def get_cached_status(self, tx_hash: str) -> Optional[Dict]:
    cache_key = f'tx_status:{tx_hash}'
    return await self.cache.get(cache_key)
```

Transaction monitoring tracks status changes and notifies callbacks.
Revert reason extraction helps debug failed transactions.

SECTION 10: COMPLETE TRANSACTION SYSTEM

Here is a complete transaction management system.

```
class TransactionSystem:
    def __init__(self, config: Dict):
        self.config = config
        self.rpc = None
        self.nonce_manager = None
        self.gas_oracle = None
        self.builder = None
        self.signer = None
        self.sender = None
        self.queue = None
        self.recovery = None
        self.monitor = None

    async def initialize(self):
        self.rpc = ResilientRPCClient(self.config["rpc_providers"])

        self.signer = TransactionSigner(self.config["private_key"])

        self.nonce_manager = NonceManager(self.rpc, self.signer.address)
        await self.nonce_manager.initialize()

        self.gas_oracle = EIP1559GasOracle(self.rpc)

        self.builder = TransactionBuilder(
            self.rpc,
            self.nonce_manager,
            self.gas_oracle
        )
        await self.builder.initialize()

        self.sender = TransactionSender(
            self.rpc,
            self.signer,
            self.nonce_manager
        )
```

```
self.queue = TransactionQueue(  
    self.builder,  
    self.sender,  
    max_pending=self.config.get("max_pending", 5)  
)
```

```
self.recovery = StuckTransactionRecovery(  
    self.rpc,  
    self.sender,  
    self.gas_oracle  
)
```

```
from indexer.cache import RedisCache  
cache = RedisCache(self.config.get("redis_url", "redis://  
localhost:6379"))  
self.monitor = TransactionMonitor(self.rpc, cache)
```

```
asyncio.create_task(self.queue.start())  
asyncio.create_task(self._recovery_loop())  
asyncio.create_task(self._nonce_sync_loop())
```

```
async def _recovery_loop(self):  
    while True:  
        try:  
            stuck = await self.recovery.find_stuck_transactions(  
                self.signer.address,  
                age_threshold=300  
            )
```

```
        if stuck:  
            print(f"Found {len(stuck)} stuck transactions")  
            results = await self.recovery.recover_all_stuck(  
                self.signer.address,  
                strategy="speed_up"  
            )  
            for result in results:  
                print(f"Recovery result: {result}")
```

```
        except Exception as e:  
            print(f"Recovery loop error: {e}")
```

```
await asyncio.sleep(60)
```

```
async def _nonce_sync_loop(self):  
while True:  
try:  
await self.nonce_manager.sync_with_chain()  
except Exception as e:  
print(f'Nonce sync error: {e}')
```

```
await asyncio.sleep(30)
```

```
async def send(  
self,  
to: str,  
value: int = 0,  
data: str = "0x",  
strategy: GasStrategy = GasStrategy.STANDARD,  
wait: bool = True  
) -> Dict:
```

```
params = TransactionParams(to=to, value=value, data=data)  
tx = await self.builder.build_transaction(params, strategy)
```

```
if wait:  
result = await self.sender.send_and_wait(tx)  
return result  
else:  
tx_hash = await self.sender.send_transaction(tx)  
return {"tx_hash": tx_hash, "status": "pending"}
```

```
async def queue_transaction(  
self,  
to: str,  
value: int = 0,  
data: str = "0x",  
strategy: GasStrategy = GasStrategy.STANDARD,  
callback: Callable = None  
) -> str:
```



```
params = TransactionParams(to=to, value=value, data=data)
tx_id = await self.queue.add(params, strategy, callback)
return tx_id
```

```
async def transfer_eth(
    self,
    to: str,
    amount_eth: float,
    strategy: GasStrategy = GasStrategy.STANDARD
) -> Dict:
```

```
value = int(amount_eth * 1e18)
return await self.send(to, value=value, strategy=strategy)
```

```
async def transfer_erc20(
    self,
    token: str,
    to: str,
    amount: int,
    strategy: GasStrategy = GasStrategy.STANDARD
) -> Dict:
```

```
tx = await self.builder.build_erc20_transfer(token, to, amount,
strategy)
return await self.sender.send_and_wait(tx)
```

```
async def call_contract(
    self,
    contract: str,
    function_sig: str,
    args: List,
    value: int = 0,
    strategy: GasStrategy = GasStrategy.STANDARD
) -> Dict:
```

```
tx = await self.builder.build_contract_call(
contract, function_sig, args, value, strategy
)
return await self.sender.send_and_wait(tx)
```

```
async def get_status(self) -> Dict:
    balance = await self.rpc.call(
        "eth_getBalance",
        [self.signer.address, "latest"]
    )
```

```
    gas_price = await
    self.gas_oracle.get_gas_price(GasStrategy.STANDARD)
```

```
    return {
        "address": self.signer.address,
        "balance_wei": int(balance, 16),
        "balance_eth": int(balance, 16) / 1e18,
        "pending_transactions": self.nonce_manager.get_pending_count(),
        "queue_stats": self.queue.get_queue_stats(),
        "gas_price_gwei": gas_price / 1e9,
        "gas_trend": self.gas_oracle.get_price_trend()
    }
```

```
async def shutdown(self):
    await self.queue.stop()
    await self.rpc.close()
```

```
async def main():
    config = {
        "rpc_providers": [
            {"name": "primary", "url": os.environ["RPC_URL"], "priority": 0}
        ],
        "private_key": os.environ["PRIVATE_KEY"],
        "redis_url": "redis://localhost:6379",
        "max_pending": 5
    }
```

```
    system = TransactionSystem(config)
    await system.initialize()
```

```
    status = await system.get_status()
    print(f"Wallet status: {status}")
```

```

result = await system.transfer_eth(
    "0x742d35Cc6634C0532925a3b844Bc9e7595f3bC28",
    0.01,
    GasStrategy.STANDARD
)
print(f"Transfer result: {result}")

await system.shutdown()

if __name__ == "__main__":
    asyncio.run(main())

```

This chapter covered comprehensive transaction management including nonce tracking with gap detection, gas price strategies for different speed requirements, transaction building with encoding, signing and broadcasting, transaction queuing for high volume, stuck transaction recovery, multi-wallet management, and transaction monitoring. Reliable transaction sending is the foundation of any blockchain application that writes to the chain.

CHAPTER 5: MONITORING AND ALERTING

You cannot fix what you cannot see. Production blockchain applications generate vast amounts of data. RPC calls, transaction statuses, cache hit rates, error rates, gas prices, wallet balances. Without monitoring, problems remain invisible until users complain. This chapter builds comprehensive observability for blockchain systems.

SECTION 1: WHAT TO MONITOR

Blockchain applications have unique monitoring requirements beyond traditional web services.

Infrastructure metrics include RPC node health and response times, database connection pool usage, cache hit rates and memory usage, CPU and memory consumption, disk I/O for indexed data, and network latency to providers.

Blockchain-specific metrics include current block number and sync

status, gas prices and trends, transaction confirmation times, wallet balances, pending transaction counts, chain reorganization frequency, and event indexing lag.

Application metrics include API request rates and latencies, error rates by type, transaction success rates, queue depths and processing times, and user activity patterns.

Business metrics include transaction volume, active wallets, NFT mints and transfers, token transfers, and revenue from fees.

SECTION 2: METRICS COLLECTION WITH PROMETHEUS

Prometheus is the standard for metrics collection. It scrapes metrics from your applications over HTTP.

```
from prometheus_client import Counter, Gauge, Histogram,  
Summary, start_http_server  
from functools import wraps  
import time
```

```
rpc_requests_total = Counter(  
    'blockchain_rpc_requests_total',  
    'Total RPC requests made',  
    ['method', 'provider', 'status']  
)
```

```
rpc_request_duration = Histogram(  
    'blockchain_rpc_request_duration_seconds',  
    'RPC request duration in seconds',  
    ['method', 'provider'],  
    buckets=[0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0, 10.0]  
)
```

```
current_block_number = Gauge(  
    'blockchain_current_block_number',  
    'Current block number',  
    ['chain']  
)
```

```
gas_price_gwei = Gauge(  
    'blockchain_gas_price_gwei',  
    'Current gas price in gwei',  
    ['chain', 'speed']  
)
```

```
wallet_balance_wei = Gauge(  
    'blockchain_wallet_balance_wei',  
    'Wallet balance in wei',  
    ['address', 'chain']  
)
```

```
pending_transactions = Gauge(  
    'blockchain_pending_transactions',  
    'Number of pending transactions',  
    ['wallet']  
)
```

```
transaction_confirmation_time = Histogram(  
    'blockchain_transaction_confirmation_seconds',  
    'Time to transaction confirmation',  
    ['chain'],  
    buckets=[10, 30, 60, 120, 300, 600, 1800]  
)
```

```
indexer_lag_blocks = Gauge(  
    'blockchain_indexer_lag_blocks',  
    'Number of blocks behind chain head',  
    ['indexer']  
)
```

```
cache_hits_total = Counter(  
    'blockchain_cache_hits_total',  
    'Cache hit count',  
    ['cache_layer', 'cache_type']  
)
```

```
cache_misses_total = Counter(  
    'blockchain_cache_misses_total',
```

```
'Cache miss count',  
['cache_layer', 'cache_type']  
)
```

```
api_requests_total = Counter(  
    'blockchain_api_requests_total',  
    'API request count',  
    ['endpoint', 'method', 'status']  
)
```

```
api_request_duration = Histogram(  
    'blockchain_api_request_duration_seconds',  
    'API request duration',  
    ['endpoint', 'method'],  
    buckets=[0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5]  
)
```

```
class MetricsCollector:  
    def __init__(self, chain: str = "ethereum"):  
        self.chain = chain  
  
    def record_rpc_request(  
        self,  
        method: str,  
        provider: str,  
        duration: float,  
        success: bool  
    ):  
        status = "success" if success else "error"  
        rpc_requests_total.labels(method=method, provider=provider,  
            status=status).inc()  
        rpc_request_duration.labels(method=method,  
            provider=provider).observe(duration)  
  
    def update_block_number(self, block_number: int):  
        current_block_number.labels(chain=self.chain).set(block_number)  
  
    def update_gas_price(self, price_gwei: float, speed: str =  
        "standard"):
```

```
gas_price_gwei.labels(chain = self.chain,  
speed = speed).set(price_gwei)
```

```
def update_wallet_balance(self, address: str, balance_wei: int):  
    wallet_balance_wei.labels(address = address,  
chain = self.chain).set(balance_wei)
```

```
def update_pending_transactions(self, wallet: str, count: int):  
    pending_transactions.labels(wallet = wallet).set(count)
```

```
def record_confirmation_time(self, seconds: float):  
    transaction_confirmation_time.labels(chain = self.chain).observe(seconds)
```

```
def update_indexer_lag(self, indexer: str, lag_blocks: int):  
    indexer_lag_blocks.labels(indexer = indexer).set(lag_blocks)
```

```
def record_cache_hit(self, layer: str, cache_type: str):  
    cache_hits_total.labels(cache_layer = layer,  
cache_type = cache_type).inc()
```

```
def record_cache_miss(self, layer: str, cache_type: str):  
    cache_misses_total.labels(cache_layer = layer,  
cache_type = cache_type).inc()
```

```
def record_api_request(  
    self,  
    endpoint: str,  
    method: str,  
    status: int,  
    duration: float  
):  
    api_requests_total.labels(  
        endpoint = endpoint, method = method, status = str(status)  
    ).inc()  
    api_request_duration.labels(endpoint = endpoint,  
method = method).observe(duration)
```

```
def track_rpc_metrics(method: str):  
    def decorator(func):
```

```

@wraps(func)
async def wrapper(self, *args, **kwargs):
    provider = getattr(self, 'current_provider', 'unknown')
    start = time.time()
    success = True

    try:
        result = await func(self, *args, **kwargs)
        return result
    except Exception as e:
        success = False
        raise
    finally:
        duration = time.time() - start
        self.metrics.record_rpc_request(method, provider, duration, success)

    return wrapper
return decorator

```

```

def start_metrics_server(port: int = 8000):
    start_http_server(port)
    print(f"Metrics server started on port {port}")

```

The metrics collector tracks all important blockchain metrics. Decorators make it easy to instrument RPC calls.

SECTION 3: STRUCTURED LOGGING

Logs provide context that metrics cannot. Structured logging makes logs searchable and analyzable.

```

import logging
import json
from datetime import datetime
from typing import Any, Dict
import sys
import traceback

```



```

class StructuredFormatter(logging.Formatter):
    def format(self, record: logging.LogRecord) -> str:
        log_entry = {
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
            "module": record.module,
            "function": record.funcName,
            "line": record.lineno
        }

        if hasattr(record, 'extra_fields'):
            log_entry.update(record.extra_fields)

        if record.exc_info:
            log_entry["exception"] = {
                "type": record.exc_info[0].__name__,
                "message": str(record.exc_info[1]),
                "traceback": traceback.format_exception(*record.exc_info)
            }

        return json.dumps(log_entry)

```

```

class BlockchainLogger:
    def __init__(self, name: str, level: int = logging.INFO):
        self.logger = logging.getLogger(name)
        self.logger.setLevel(level)

        if not self.logger.handlers:
            handler = logging.StreamHandler(sys.stdout)
            handler.setFormatter(StructuredFormatter())
            self.logger.addHandler(handler)

        self.default_fields: Dict[str, Any] = {}

        def set_default_field(self, key: str, value: Any):
            self.default_fields[key] = value

```

```
def _log(self, level: int, message: str, **kwargs):
    extra = {"extra_fields": {**self.default_fields, **kwargs}}
    self.logger.log(level, message, extra = extra)
```

```
def debug(self, message: str, **kwargs):
    self._log(logging.DEBUG, message, **kwargs)
```

```
def info(self, message: str, **kwargs):
    self._log(logging.INFO, message, **kwargs)
```

```
def warning(self, message: str, **kwargs):
    self._log(logging.WARNING, message, **kwargs)
```

```
def error(self, message: str, **kwargs):
    self._log(logging.ERROR, message, **kwargs)
```

```
def exception(self, message: str, **kwargs):
    extra = {"extra_fields": {**self.default_fields, **kwargs}}
    self.logger.exception(message, extra = extra)
```

```
def rpc_call(
    self,
    method: str,
    provider: str,
    duration_ms: float,
    success: bool,
    **kwargs
):
    self.info(
        f"RPC call: {method}",
        event_type="rpc_call",
        rpc_method=method,
        provider=provider,
        duration_ms=round(duration_ms, 2),
        success=success,
        **kwargs
    )
```

```
def transaction_sent(
    self,
```

```
tx_hash: str,  
from_address: str,  
to_address: str,  
value: int,  
gas_price: int,  
**kwargs  
):  
    self.info(  
        f"Transaction sent: {tx_hash}",  
        event_type="transaction_sent",  
        tx_hash=tx_hash,  
        from_address=from_address,  
        to_address=to_address,  
        value_wei=value,  
        gas_price_gwei=gas_price / 1e9,  
        **kwargs  
    )
```

```
def transaction_confirmed(  
    self,  
    tx_hash: str,  
    block_number: int,  
    gas_used: int,  
    confirmation_time_seconds: float,  
    **kwargs  
):  
    self.info(  
        f"Transaction confirmed: {tx_hash}",  
        event_type="transaction_confirmed",  
        tx_hash=tx_hash,  
        block_number=block_number,  
        gas_used=gas_used,  
        confirmation_time_seconds=round(confirmation_time_seconds, 2),  
        **kwargs  
    )
```

```
def transaction_failed(  
    self,  
    tx_hash: str,  
    error: str,
```

```
**kwargs
):
self.error(
f"Transaction failed: {tx_hash}",
event_type="transaction_failed",
tx_hash=tx_hash,
error=error,
**kwargs
)
```

```
def indexer_progress(
self,
indexer_name: str,
current_block: int,
target_block: int,
events_processed: int,
**kwargs
):
lag = target_block - current_block
progress = (current_block / target_block) * 100 if target_block > 0
else 0
```

```
self.info(
f"Indexer progress: {indexer_name}",
event_type="indexer_progress",
indexer=indexer_name,
current_block=current_block,
target_block=target_block,
lag_blocks=lag,
progress_percent=round(progress, 2),
events_processed=events_processed,
**kwargs
)
```

```
def alert(
self,
alert_name: str,
severity: str,
message: str,
**kwargs
```

```

):
    level = logging.CRITICAL if severity == "critical" else
logging.WARNING

```

```

self._log(
    level,
    f'ALERT [{severity}] {alert_name}: {message}',
    event_type="alert",
    alert_name=alert_name,
    severity=severity,
    **kwargs
)

```

Structured logging produces JSON output that log aggregation systems can parse and index.

SECTION 4: LOG AGGREGATION WITH ELASTICSEARCH

Centralize logs for searching and analysis.

```

from datetime import datetime
from elasticsearch import AsyncElasticsearch
from typing import List, Dict, Any

```

```

class ElasticsearchLogSink:
    def __init__(self, hosts: List[str], index_prefix: str = "blockchain-
logs"):
        self.client = AsyncElasticsearch(hosts)
        self.index_prefix = index_prefix
        self.buffer: List[Dict] = []
        self.buffer_size = 100
        self.flush_interval = 5.0

```

```

    def _get_index_name(self) -> str:
        date = datetime.utcnow().strftime("%Y.%m.%d")
        return f"{self.index_prefix}-{date}"

```

```

    async def log(self, entry: Dict):
        entry["@timestamp"] = datetime.utcnow().isoformat() + "Z"

```

```

self.buffer.append(entry)

if len(self.buffer) >= self.buffer_size:
    await self.flush()

async def flush(self):
    if not self.buffer:
        return

    entries = self.buffer
    self.buffer = []

    actions = []
    index_name = self._get_index_name()

    for entry in entries:
        actions.append({"index": {"_index": index_name}})
        actions.append(entry)

    try:
        await self.client.bulk(operations=actions)
    except Exception as e:
        print(f"Failed to send logs to Elasticsearch: {e}")
        self.buffer.extend(entries)

    async def search(
        self,
        query: str,
        start_time: datetime = None,
        end_time: datetime = None,
        size: int = 100
    ) -> List[Dict]:

        must_clauses = [
            {"query_string": {"query": query}}
        ]

        if start_time or end_time:
            time_range = {}
            if start_time:

```

```

time_range["gte"] = start_time.isoformat()
if end_time:
    time_range["lte"] = end_time.isoformat()
must_clauses.append({"range": {"@timestamp": time_range}})

```

```

search_body = {
    "query": {"bool": {"must": must_clauses}},
    "sort": [{"@timestamp": "desc"}],
    "size": size
}

```

```

result = await self.client.search(
    index=f'{self.index_prefix}-*',
    body=search_body
)

```

```

return [hit["_source"]] for hit in result["hits"]["hits"]

```

```

async def get_error_summary(
    self,
    hours: int = 24
) -> Dict[str, int]:

```

```

start_time = datetime.utcnow() - timedelta(hours=hours)

```

```

search_body = {
    "query": {
        "bool": {
            "must": [
                {"term": {"level": "ERROR"}},
                {"range": {"@timestamp": {"gte": start_time.isoformat()}}}
            ]
        }
    },
    "aggs": {
        "error_types": {
            "terms": {"field": "event_type.keyword", "size": 20}
        }
    },
    "size": 0
}

```

```
}
```

```
result = await self.client.search(  
    index=f'{self.index_prefix}-*',  
    body=search_body  
)
```

```
buckets = result["aggregations"]["error_types"]["buckets"]  
return {b["key"]: b["doc_count"] for b in buckets}
```

```
async def close(self):  
    await self.flush()  
    await self.client.close()
```

```
class LoggingHandler(logging.Handler):  
    def __init__(self, es_sink: ElasticsearchLogSink):  
        super().__init__()  
        self.sink = es_sink  
        self.loop = None
```

```
    def emit(self, record: logging.LogRecord):  
        try:  
            entry = json.loads(self.format(record))
```

```
            if self.loop is None:  
                import asyncio  
                self.loop = asyncio.get_event_loop()
```

```
            self.loop.create_task(self.sink.log(entry))
```

```
        except Exception:  
            self.handleError(record)
```

Elasticsearch stores logs for long-term retention and enables complex queries across all your services.

SECTION 5: ALERTING RULES

Define conditions that trigger alerts when something goes wrong.

```
from dataclasses import dataclass
from typing import Callable, Optional, List, Dict, Any
from enum import Enum
import asyncio
```

```
class AlertSeverity(Enum):
    INFO = "info"
    WARNING = "warning"
    CRITICAL = "critical"
```

```
@dataclass
class AlertRule:
    name: str
    description: str
    condition: Callable[[], bool]
    severity: AlertSeverity
    cooldown_seconds: int = 300
    last_triggered: float = 0
```

```
@dataclass
class Alert:
    rule_name: str
    severity: AlertSeverity
    message: str
    timestamp: float
    context: Dict[str, Any]
```

```
class AlertManager:
    def __init__(self, logger: BlockchainLogger):
        self.logger = logger
        self.rules: Dict[str, AlertRule] = {}
        self.active_alerts: Dict[str, Alert] = {}
        self.handlers: List[Callable] = []
        self.running = False
```

```
    def register_rule(self, rule: AlertRule):
        self.rules[rule.name] = rule
```

```

def register_handler(self, handler: Callable):
    self.handlers.append(handler)

async def check_rules(self):
    now = time.time()

    for rule in self.rules.values():
        if now - rule.last_triggered < rule.cooldown_seconds:
            continue

        try:
            if asyncio.iscoroutinefunction(rule.condition):
                triggered = await rule.condition()
            else:
                triggered = rule.condition()

            if triggered:
                rule.last_triggered = now

                alert = Alert(
                    rule_name=rule.name,
                    severity=rule.severity,
                    message=rule.description,
                    timestamp=now,
                    context={}
                )

                await self._fire_alert(alert)

        except Exception as e:
            self.logger.error(
                f'Error checking alert rule {rule.name}',
                error=str(e)
            )

    async def _fire_alert(self, alert: Alert):
        self.active_alerts[alert.rule_name] = alert

    self.logger.alert(
        alert.rule_name,

```

```
    alert.severity.value,  
    alert.message,  
    **alert.context  
)
```

```
for handler in self.handlers:  
    try:  
        if asyncio.iscoroutinefunction(handler):  
            await handler(alert)  
        else:  
            handler(alert)  
    except Exception as e:  
        self.logger.error(  
            f'Alert handler error',  
            error = str(e)  
)
```

```
def resolve_alert(self, rule_name: str):  
    if rule_name in self.active_alerts:  
        del self.active_alerts[rule_name]  
        self.logger.info(  
            f'Alert resolved: {rule_name}',  
            event_type = "alert_resolved"  
)
```

```
async def start(self, check_interval: int = 30):  
    self.running = True
```

```
while self.running:  
    await self.check_rules()  
    await asyncio.sleep(check_interval)
```

```
def stop(self):  
    self.running = False
```

```
class BlockchainAlerts:  
    def __init__(  
        self,  
        rpc_client,
```

```

nonce_manager,
gas_oracle,
metrics_collector,
alert_manager: AlertManager
):
self.rpc = rpc_client
self.nonce_manager = nonce_manager
self.gas_oracle = gas_oracle
self.metrics = metrics_collector
self.alerts = alert_manager
self.thresholds = {
"low_balance_eth": 0.1,
"high_gas_gwei": 100,
"max_pending_transactions": 10,
"max_indexer_lag": 100,
"max_rpc_latency_ms": 1000,
"min_rpc_success_rate": 0.95
}

```

```

def configure_thresholds(self, thresholds: Dict):
self.thresholds.update(thresholds)

```

```

async def setup_rules(self, wallet_address: str):
self.alerts.register_rule(AlertRule(
name="low_wallet_balance",
description=f"Wallet balance below
{self.thresholds['low_balance_eth']} ETH",
condition=lambda: self._check_low_balance(wallet_address),
severity=AlertSeverity.CRITICAL,
cooldown_seconds=600
))

```

```

self.alerts.register_rule(AlertRule(
name="high_gas_price",
description=f"Gas price above {self.thresholds['high_gas_gwei']}
gwei",
condition=self._check_high_gas,
severity=AlertSeverity.WARNING,
cooldown_seconds=300
))

```

```
self.alerts.register_rule(AlertRule(
    name="stuck_transactions",
    description="Transactions stuck in pending state",
    condition=self._check_stuck_transactions,
    severity=AlertSeverity.WARNING,
    cooldown_seconds=300
))
```

```
self.alerts.register_rule(AlertRule(
    name="rpc_degraded",
    description="RPC response times degraded",
    condition=self._check_rpc_latency,
    severity=AlertSeverity.WARNING,
    cooldown_seconds=180
))
```

```
self.alerts.register_rule(AlertRule(
    name="indexer_behind",
    description=f"Indexer more than
{self.thresholds['max_indexer_lag']} blocks behind",
    condition=self._check_indexer_lag,
    severity=AlertSeverity.WARNING,
    cooldown_seconds=300
))
```

```
async def _check_low_balance(self, address: str) -> bool:
    try:
        balance = await self.rpc.call("eth_getBalance", [address, "latest"])
        balance_eth = int(balance, 16) / 1e18
        return balance_eth < self.thresholds["low_balance_eth"]
    except Exception:
        return False
```

```
async def _check_high_gas(self) -> bool:
    try:
        gas_price = await self.gas_oracle.get_gas_price()
        gas_gwei = gas_price / 1e9
        return gas_gwei > self.thresholds["high_gas_gwei"]
    except Exception:
```

```
return False
```

```
async def _check_stuck_transactions(self) -> bool:  
    pending = self.nonce_manager.get_pending_count()  
    return pending > self.thresholds["max_pending_transactions"]
```

```
async def _check_rpc_latency(self) -> bool:  
    return False
```

```
async def _check_indexer_lag(self) -> bool:  
    return False
```

SECTION 6: NOTIFICATION CHANNELS

Send alerts through multiple channels to ensure visibility.

```
import aiohttp  
from abc import ABC, abstractmethod
```

```
class NotificationChannel(ABC):  
    @abstractmethod  
    async def send(self, alert: Alert) -> bool:  
        pass
```

```
class SlackNotifier(NotificationChannel):  
    def __init__(self, webhook_url: str, channel: str = None):  
        self.webhook_url = webhook_url  
        self.channel = channel
```

```
    async def send(self, alert: Alert) -> bool:  
        color_map = {  
            AlertSeverity.INFO: "#36a64f",  
            AlertSeverity.WARNING: "#ffcc00",  
            AlertSeverity.CRITICAL: "#ff0000"  
        }
```

```
        payload = {  
            "attachments": [{  
                "color": color_map[alert.severity],
```

```

"title": f"[{alert.severity.value.upper()}] {alert.rule_name}",
"text": alert.message,
"fields": [
{
"title": key,
"value": str(value),
"short": True
}
for key, value in alert.context.items()
],
"ts": int(alert.timestamp)
}]
}

```

```

if self.channel:
    payload["channel"] = self.channel

```

```

try:
    async with aiohttp.ClientSession() as session:
        async with session.post(self.webhook_url, json=payload) as
response:
    return response.status == 200
except Exception:
    return False

```

```

class PagerDutyNotifier(NotificationChannel):
    def __init__(self, routing_key: str):
        self.routing_key = routing_key
        self.api_url = "https://events.pagerduty.com/v2/enqueue"

```

```

async def send(self, alert: Alert) -> bool:
    severity_map = {
        AlertSeverity.INFO: "info",
        AlertSeverity.WARNING: "warning",
        AlertSeverity.CRITICAL: "critical"
    }

```

```

payload = {
    "routing_key": self.routing_key,

```

```

"event_action": "trigger",
"dedup_key": alert.rule_name,
"payload": {
"summary": f"{alert.rule_name}: {alert.message}",
"severity": severity_map[alert.severity],
"source": "blockchain-monitor",
"timestamp": datetime.fromtimestamp(alert.timestamp).isoformat(),
"custom_details": alert.context
}
}

```

```

try:
    async with aiohttp.ClientSession() as session:
        async with session.post(self.api_url, json = payload) as response:
            return response.status == 202
except Exception:
    return False

```

```

async def resolve(self, rule_name: str) -> bool:
    payload = {
        "routing_key": self.routing_key,
        "event_action": "resolve",
        "dedup_key": rule_name
    }

```

```

try:
    async with aiohttp.ClientSession() as session:
        async with session.post(self.api_url, json = payload) as response:
            return response.status == 202
except Exception:
    return False

```

```

class TelegramNotifier(NotificationChannel):
    def __init__(self, bot_token: str, chat_id: str):
        self.bot_token = bot_token
        self.chat_id = chat_id
        self.api_url = f"https://api.telegram.org/bot{bot_token}/
sendMessage"

```



```
async def send(self, alert: Alert) -> bool:
```

```
    emoji_map = {  
        AlertSeverity.INFO: "i👉",  
        AlertSeverity.WARNING: "⚠️👉",  
        AlertSeverity.CRITICAL: "🔴👉"  
    }
```

```
    message = f"{emoji_map[alert.severity]} *{alert.rule_name}*\n\n"  
    message += f"{alert.message}\n\n"
```

```
    if alert.context:
```

```
        message += "*Details:*\n"  
        for key, value in alert.context.items():  
            message += f"• {key}: `{value}`\n"
```

```
    payload = {  
        "chat_id": self.chat_id,  
        "text": message,  
        "parse_mode": "Markdown"  
    }
```

```
    try:
```

```
        async with aiohttp.ClientSession() as session:  
            async with session.post(self.api_url, json=payload) as response:  
                return response.status == 200  
        except Exception:  
            return False
```

```
class DiscordNotifier(NotificationChannel):
```

```
    def __init__(self, webhook_url: str):  
        self.webhook_url = webhook_url
```

```
    async def send(self, alert: Alert) -> bool:
```

```
        color_map = {  
            AlertSeverity.INFO: 0x36a64f,  
            AlertSeverity.WARNING: 0xffcc00,  
            AlertSeverity.CRITICAL: 0xff0000  
        }
```

```

embed = {
    "title": f"[{alert.severity.value.upper()}] {alert.rule_name}",
    "description": alert.message,
    "color": color_map[alert.severity],
    "fields": [
        {"name": key, "value": str(value), "inline": True}
        for key, value in alert.context.items()
    ],
    "timestamp": datetime.fromtimestamp(alert.timestamp).isoformat()
}

```

```

payload = {"embeds": [embed]}

```

```

try:
    async with aiohttp.ClientSession() as session:
        async with session.post(self.webhook_url, json=payload) as
response:
    return response.status == 204
except Exception:
    return False

```

```

class MultiChannelNotifier:
    def __init__(self):
        self.channels: List[NotificationChannel] = []
        self.severity_routing: Dict[AlertSeverity, List[int]] = {
            AlertSeverity.INFO: [],
            AlertSeverity.WARNING: [],
            AlertSeverity.CRITICAL: []
        }

```

```

    def add_channel(
        self,
        channel: NotificationChannel,
        severities: List[AlertSeverity] = None
    ):
        channel_index = len(self.channels)
        self.channels.append(channel)

```

```

    if severities is None:

```

```

severities = list(AlertSeverity)

for severity in severities:
    self.severity_routing[severity].append(channel_index)

async def notify(self, alert: Alert):
    channel_indices = self.severity_routing.get(alert.severity, [])

    tasks = []
    for idx in channel_indices:
        tasks.append(self.channels[idx].send(alert))

    if tasks:
        results = await asyncio.gather(*tasks, return_exceptions=True)
        return all(r is True for r in results)

    return False

```

Multiple notification channels ensure alerts reach the right people through their preferred medium.

SECTION 7: GRAFANA DASHBOARDS

Visualize metrics with Grafana dashboards. Here is a dashboard definition.

```

{
  "title": "Blockchain Infrastructure",
  "uid": "blockchain-infra",
  "panels": [
    {
      "title": "RPC Request Rate",
      "type": "graph",
      "gridPos": {"x": 0, "y": 0, "w": 12, "h": 8},
      "targets": [
        {
          "expr": "rate(blockchain_rpc_requests_total[5m])",
          "legendFormat": "{{method}} - {{provider}}"
        }
      ]
    }
  ]
}

```

```

]
},
{
  "title": "RPC Latency (p95)",
  "type": "graph",
  "gridPos": {"x": 12, "y": 0, "w": 12, "h": 8},
  "targets": [
    {
      "expr": "histogram_quantile(0.95,
rate(blockchain_rpc_request_duration_seconds_bucket[5m]))",
      "legendFormat": "{{method}}"
    }
  ]
},
{
  "title": "Current Block Number",
  "type": "stat",
  "gridPos": {"x": 0, "y": 8, "w": 6, "h": 4},
  "targets": [
    {
      "expr": "blockchain_current_block_number",
      "legendFormat": "{{chain}}"
    }
  ]
},
{
  "title": "Gas Price (gwei)",
  "type": "gauge",
  "gridPos": {"x": 6, "y": 8, "w": 6, "h": 4},
  "targets": [
    {
      "expr": "blockchain_gas_price_gwei{speed=\"standard\"}",
      "legendFormat": "{{chain}}"
    }
  ]
},
"fieldConfig": {
  "defaults": {
    "thresholds": {
      "steps": [
        {"value": 0, "color": "green"},

```

```

{"value": 50, "color": "yellow"},
{"value": 100, "color": "red"}
]
}
}
}
},
{
  "title": "Wallet Balance (ETH)",
  "type": "stat",
  "gridPos": {"x": 12, "y": 8, "w": 6, "h": 4},
  "targets": [
    {
      "expr": "blockchain_wallet_balance_wei / 1e18",
      "legendFormat": "{{address}}"
    }
  ],
},
{
  "title": "Pending Transactions",
  "type": "stat",
  "gridPos": {"x": 18, "y": 8, "w": 6, "h": 4},
  "targets": [
    {
      "expr": "blockchain_pending_transactions",
      "legendFormat": "{{wallet}}"
    }
  ],
},
{
  "title": "Transaction Confirmation Time",
  "type": "heatmap",
  "gridPos": {"x": 0, "y": 12, "w": 12, "h": 8},
  "targets": [
    {
      "expr":
"rate(blockchain_transaction_confirmation_seconds_bucket[5m])",
      "format": "heatmap"
    }
  ]
}

```

```

},
{
  "title": "Indexer Lag",
  "type": "graph",
  "gridPos": {"x": 12, "y": 12, "w": 12, "h": 8},
  "targets": [
    {
      "expr": "blockchain_indexer_lag_blocks",
      "legendFormat": "{{indexer}}"
    }
  ],
},
{
  "title": "Cache Hit Rate",
  "type": "gauge",
  "gridPos": {"x": 0, "y": 20, "w": 8, "h": 6},
  "targets": [
    {
      "expr": "sum(rate(blockchain_cache_hits_total[5m])) /
(sum(rate(blockchain_cache_hits_total[5m])) +
sum(rate(blockchain_cache_misses_total[5m])))",
      "legendFormat": "Hit Rate"
    }
  ],
},
{
  "title": "API Request Rate",
  "type": "graph",
  "gridPos": {"x": 8, "y": 20, "w": 8, "h": 6},
  "targets": [
    {
      "expr": "rate(blockchain_api_requests_total[5m])",
      "legendFormat": "{{endpoint}} - {{status}}"
    }
  ],
},
{
  "title": "API Latency (p99)",
  "type": "graph",
  "gridPos": {"x": 16, "y": 20, "w": 8, "h": 6},

```

```

"targets": [
{
"expr": "histogram_quantile(0.99,
rate(blockchain_api_request_duration_seconds_bucket[5m]))",
"legendFormat": "{{endpoint}}"
}
]
}
]
}

```

SECTION 8: HEALTH CHECK ENDPOINTS

Expose health endpoints for load balancers and orchestration systems.

```

from aiohttp import web
from typing import Dict, List

```

```

class HealthCheckServer:
    def __init__(
        self,
        rpc_client,
        database,
        redis_cache,
        indexer=None
    ):
        self.rpc = rpc_client
        self.db = database
        self.cache = redis_cache
        self.indexer = indexer
        self.checks: Dict[str, Callable] = {}

```

```

    def register_check(self, name: str, check_func: Callable):
        self.checks[name] = check_func

```

```

    async def check_rpc(self) -> Dict:
        try:
            start = time.time()

```

```
await self.rpc.call("eth_blockNumber")
latency = (time.time() - start) * 1000
```

```
return {
    "status": "healthy",
    "latency_ms": round(latency, 2)
}
except Exception as e:
    return {
        "status": "unhealthy",
        "error": str(e)
    }
```

```
async def check_database(self) -> Dict:
    try:
        start = time.time()
        await self.db.pool.fetchval("SELECT 1")
        latency = (time.time() - start) * 1000
```

```
        pool_size = self.db.pool.get_size()
        pool_free = self.db.pool.get_idle_size()
```

```
    return {
        "status": "healthy",
        "latency_ms": round(latency, 2),
        "pool_size": pool_size,
        "pool_free": pool_free
    }
    except Exception as e:
        return {
            "status": "unhealthy",
            "error": str(e)
        }
```

```
async def check_cache(self) -> Dict:
    try:
        start = time.time()
        await self.cache.redis.ping()
        latency = (time.time() - start) * 1000
```



```
info = await self.cache.redis.info()
```

```
return {  
    "status": "healthy",  
    "latency_ms": round(latency, 2),  
    "used_memory": info.get("used_memory_human"),  
    "connected_clients": info.get("connected_clients")  
}  
except Exception as e:  
    return {  
        "status": "unhealthy",  
        "error": str(e)  
    }
```

```
async def check_indexer(self) -> Dict:  
    if not self.indexer:  
        return {"status": "not_configured"}
```

```
try:  
    current_block = await self.rpc.call("eth_blockNumber")  
    current_block = int(current_block, 16)
```

```
indexed_block = self.indexer.state.last_confirmed_block  
lag = current_block - indexed_block
```

```
return {  
    "status": "healthy" if lag < 100 else "degraded",  
    "current_block": current_block,  
    "indexed_block": indexed_block,  
    "lag_blocks": lag  
}  
except Exception as e:  
    return {  
        "status": "unhealthy",  
        "error": str(e)  
    }
```

```
async def run_all_checks(self) -> Dict:  
    results = {}
```

```
results["rpc"] = await self.check_rpc()
results["database"] = await self.check_database()
results["cache"] = await self.check_cache()
results["indexer"] = await self.check_indexer()
```

```
for name, check_func in self.checks.items():
    try:
        if asyncio.iscoroutinefunction(check_func):
            results[name] = await check_func()
        else:
            results[name] = check_func()
    except Exception as e:
        results[name] = {"status": "error", "error": str(e)}
```

```
all_healthy = all(
    r.get("status") in ["healthy", "not_configured"]
    for r in results.values()
)
```

```
return {
    "healthy": all_healthy,
    "checks": results,
    "timestamp": datetime.utcnow().isoformat()
}
```

```
async def liveness_handler(self, request: web.Request) ->
web.Response:
    return web.json_response({"status": "alive"})
```

```
async def readiness_handler(self, request: web.Request) ->
web.Response:
    results = await self.run_all_checks()
    status = 200 if results["healthy"] else 503
    return web.json_response(results, status=status)
```

```
async def health_handler(self, request: web.Request) ->
web.Response:
    results = await self.run_all_checks()
    status = 200 if results["healthy"] else 503
    return web.json_response(results, status=status)
```

```
def create_app(self) -> web.Application:
    app = web.Application()
    app.router.add_get("/livez", self.liveness_handler)
    app.router.add_get("/readyz", self.readiness_handler)
    app.router.add_get("/healthz", self.health_handler)
    return app
```

```
async def start(self, port: int = 8080):
    app = self.create_app()
    runner = web.AppRunner(app)
    await runner.setup()
    site = web.TCPSite(runner, "0.0.0.0", port)
    await site.start()
    print(f"Health check server started on port {port}")
```

SECTION 9: DISTRIBUTED TRACING

Track requests across services to identify performance bottlenecks.

```
import uuid
from contextvars import ContextVar
from typing import Optional

trace_id_var: ContextVar[Optional[str]] = ContextVar('trace_id',
default=None)
span_id_var: ContextVar[Optional[str]] = ContextVar('span_id',
default=None)

@dataclass
class Span:
    trace_id: str
    span_id: str
    parent_span_id: Optional[str]
    operation_name: str
    start_time: float
    end_time: Optional[float] = None
    tags: Dict[str, Any] = field(default_factory=dict)
    logs: List[Dict] = field(default_factory=list)
```

```

class Tracer:
    def __init__(self, service_name: str, exporter = None):
        self.service_name = service_name
        self.exporter = exporter
        self.active_spans: Dict[str, Span] = {}

    def start_trace(self) -> str:
        trace_id = uuid.uuid4().hex
        trace_id_var.set(trace_id)
        return trace_id

    def start_span(
        self,
        operation_name: str,
        tags: Dict[str, Any] = None
    ) -> Span:

        trace_id = trace_id_var.get() or self.start_trace()
        parent_span_id = span_id_var.get()
        span_id = uuid.uuid4().hex[:16]

        span = Span(
            trace_id=trace_id,
            span_id=span_id,
            parent_span_id=parent_span_id,
            operation_name=operation_name,
            start_time=time.time(),
            tags={"service": self.service_name, **(tags or {})}
        )

        span_id_var.set(span_id)
        self.active_spans[span_id] = span

    return span

    def finish_span(self, span: Span):
        span.end_time = time.time()

    if span.span_id in self.active_spans:

```

```
del self.active_spans[span.span_id]
```

```
if span.parent_span_id:  
    span_id_var.set(span.parent_span_id)  
else:  
    span_id_var.set(None)
```

```
if self.exporter:  
    self.exporter.export(span)
```

```
def add_tag(self, span: Span, key: str, value: Any):  
    span.tags[key] = value
```

```
def log(self, span: Span, message: str, **kwargs):  
    span.logs.append({  
        "timestamp": time.time(),  
        "message": message,  
        **kwargs  
    })
```

```
class TracingMiddleware:  
    def __init__(self, tracer: Tracer):  
        self.tracer = tracer
```

```
    async def _call(self, request, handler):  
        trace_id = request.headers.get("X-Trace-ID")  
        if trace_id:  
            trace_id_var.set(trace_id)
```

```
        span = self.tracer.start_span(  
            f'{request.method} {request.path}',  
            tags={  
                "http.method": request.method,  
                "http.url": str(request.url)  
            }  
        )
```

```
    try:  
        response = await handler(request)
```

```
self.tracer.add_tag(span, "http.status_code", response.status)
return response
```

```
except Exception as e:
self.tracer.add_tag(span, "error", True)
self.tracer.add_tag(span, "error.message", str(e))
raise
```

```
finally:
self.tracer.finish_span(span)
```

```
def traced(operation_name: str = None):
def decorator(func):
@wraps(func)
async def wrapper(*args, **kwargs):
tracer = args[0].tracer if hasattr(args[0], 'tracer') else None
```

```
if not tracer:
return await func(*args, **kwargs)
```

```
name = operation_name or func.__name__
span = tracer.start_span(name)
```

```
try:
result = await func(*args, **kwargs)
return result
except Exception as e:
tracer.add_tag(span, "error", True)
tracer.add_tag(span, "error.message", str(e))
raise
finally:
tracer.finish_span(span)
```

```
return wrapper
return decorator
```

Distributed tracing shows how requests flow through your system and where time is spent.

SECTION 10: COMPLETE MONITORING SETUP

Tie everything together into a production monitoring system.

```
async def setup_monitoring(config: Dict) -> Dict:
    start_metrics_server(config.get("metrics_port", 8000))

    metrics = MetricsCollector(chain = config.get("chain", "ethereum"))

    logger = BlockchainLogger("blockchain")
    logger.set_default_field("environment", config.get("environment",
"production"))
    logger.set_default_field("service", config.get("service_name",
"blockchain-app"))

    es_sink = ElasticsearchLogSink(
        hosts = config.get("elasticsearch_hosts", ["http://localhost:9200"]),
        index_prefix = "blockchain-logs"
    )

    es_handler = LoggingHandler(es_sink)
    es_handler.setFormatter(StructuredFormatter())
    logger.logger.addHandler(es_handler)

    alert_manager = AlertManager(logger)

    notifier = MultiChannelNotifier()

    if config.get("slack_webhook"):
        slack = SlackNotifier(config["slack_webhook"])
        notifier.add_channel(slack, [AlertSeverity.WARNING,
AlertSeverity.CRITICAL])

    if config.get("pagerduty_key"):
        pagerduty = PagerDutyNotifier(config["pagerduty_key"])
        notifier.add_channel(pagerduty, [AlertSeverity.CRITICAL])

    if config.get("telegram_bot_token"):
        telegram = TelegramNotifier(
```

```
config["telegram_bot_token"],
config["telegram_chat_id"]
)
notifier.add_channel(telegram)
```

```
alert_manager.register_handler(notifier.notify)
```

```
tracer = Tracer(config.get("service_name", "blockchain-app"))
```

```
health_server = HealthCheckServer(
rpc_client = config.get("rpc_client"),
database = config.get("database"),
redis_cache = config.get("redis_cache"),
indexer = config.get("indexer")
)
```

```
asyncio.create_task(alert_manager.start(check_interval = 30))
asyncio.create_task(health_server.start(port = config.get("health_port",
8080)))
```

```
logger.info("Monitoring system initialized")
```

```
return {
"metrics": metrics,
"logger": logger,
"alerts": alert_manager,
"tracer": tracer,
"health": health_server
}
```

```
async def main():
config = {
"chain": "ethereum",
"environment": "production",
"service_name": "blockchain-indexer",
"metrics_port": 8000,
"health_port": 8080,
"elasticsearch_hosts": ["http://elasticsearch:9200"],
"slack_webhook": os.environ.get("SLACK_WEBHOOK"),
```



```
"pagerduty_key": os.environ.get("PAGERDUTY_KEY"),  
"telegram_bot_token": os.environ.get("TELEGRAM_BOT_TOKEN"),  
"telegram_chat_id": os.environ.get("TELEGRAM_CHAT_ID")  
}
```

```
monitoring = await setup_monitoring(config)
```

```
monitoring["logger"].info("Application started")
```

```
while True:
```

```
    monitoring["metrics"].update_block_number(12345678)
```

```
    monitoring["metrics"].update_gas_price(50.5, "standard")
```

```
    await asyncio.sleep(10)
```

```
if __name__ == "__main__":
```

```
    asyncio.run(main())
```

This chapter covered comprehensive monitoring and alerting for blockchain applications. You learned Prometheus metrics collection for quantitative data, structured logging for context and debugging, Elasticsearch for log aggregation and search, alert rules that detect problems automatically, notification channels for Slack, PagerDuty, Telegram and Discord, Grafana dashboards for visualization, health check endpoints for orchestration, and distributed tracing for request flow analysis. With proper monitoring, you see problems before users do.

CHAPTER 6: HIGH AVAILABILITY ARCHITECTURE

Availability is the percentage of time your system operates correctly. Five nines availability means 99.999 percent uptime or about 5 minutes of downtime per year. Achieving high availability requires designing systems that survive component failures. This chapter covers patterns for building blockchain applications that stay up.

SECTION 1: AVAILABILITY FUNDAMENTALS

Availability depends on two factors. Mean Time Between Failures

(MTBF) measures how often things break. Mean Time To Recovery (MTTR) measures how quickly you fix them. Availability equals MTBF divided by the sum of MTBF and MTTR.

You can improve availability by increasing MTBF through better components, redundancy, and preventive maintenance. You can also improve availability by decreasing MTTR through automated failover, quick deployment, and good monitoring.

For blockchain applications, you face multiple failure points. RPC providers can have outages. Databases can crash. Cache servers can fail. Your own code can have bugs. Each failure point reduces overall availability.

If each component has 99 percent availability and you have 5 components in series, overall availability is 0.99 to the power of 5, which equals about 95 percent. That is over 18 days of downtime per year.

The solution is redundancy. When one component fails, another takes over.

SECTION 2: STATELESS SERVICE DESIGN

Stateless services are easier to scale and more resilient. Any instance can handle any request. Failed instances can be replaced instantly.

```
from abc import ABC, abstractmethod
from typing import Dict, Any, Optional
import asyncio
```

```
class StatelessService(ABC):
    def __init__(self, config: Dict):
        self.config = config
        self.instance_id = self._generate_instance_id()
        self.healthy = True
```

```
    def _generate_instance_id(self) -> str:
```

```
import uuid
return uuid.uuid4().hex[:8]
```

```
@abstractmethod
async def handle_request(self, request: Dict) -> Dict:
    pass
```

```
async def health_check(self) -> bool:
    return self.healthy
```

```
class BlockchainQueryService(StatelessService):
    def __init__(self, config: Dict):
        super().__init__(config)
        self.rpc_manager = None
        self.cache = None
        self.db_pool = None
```

```
    async def initialize(self):
        from infrastructure.rpc import RPCProviderManager,
        ResilientRPCClient
        from infrastructure.cache import TieredCache
        import asyncpg
```

```
        providers = [
            {"name": p["name"], "url": p["url"], "priority": i}
            for i, p in enumerate(self.config["rpc_providers"])
        ]
        self.rpc_manager = ResilientRPCClient(providers)
```

```
        self.db_pool = await asyncpg.create_pool(
            self.config["database_url"],
            min_size=5,
            max_size=20
        )
```

```
        from redis.asyncio import Redis
        redis = Redis.from_url(self.config["redis_url"])
```

```
        from infrastructure.cache import BlockAwareCache, RedisCache
```

```
memory = BlockAwareCache()
redis_cache = RedisCache(self.config["redis_url"])
```

```
self.cache = TieredCache(memory, redis_cache, self.rpc_manager,
self.db_pool)
```

```
async def handle_request(self, request: Dict) -> Dict:
    action = request.get("action")
```

```
    if action == "get_balance":
        return await self._get_balance(request)
    elif action == "get_transaction":
        return await self._get_transaction(request)
    elif action == "get_nft_owner":
        return await self._get_nft_owner(request)
    else:
        return {"error": "Unknown action"}
```

```
    async def _get_balance(self, request: Dict) -> Dict:
        address = request["address"]
        token = request.get("token")
```

```
        balance = await self.cache.get_balance(address, token)
```

```
        return {"address": address, "balance": balance}
```

```
    async def _get_transaction(self, request: Dict) -> Dict:
        tx_hash = request["tx_hash"]
```

```
        tx = await self.rpc_manager.call(
            "eth_getTransactionByHash",
            [tx_hash]
        )
```

```
        return {"transaction": tx}
```

```
    async def _get_nft_owner(self, request: Dict) -> Dict:
        contract = request["contract"]
        token_id = request["token_id"]
```

```
owner = await self.cache.get_nft_owner(contract, token_id)
```

```
return {"owner": owner}
```

```
async def shutdown(self):  
    if self.db_pool:  
        await self.db_pool.close()
```

Stateless design means the service has no local state that must be preserved. Everything is stored in external systems that handle their own redundancy.

SECTION 3: DATABASE REPLICATION

PostgreSQL replication provides read scalability and failover capability.

Here is a Docker Compose setup for PostgreSQL with streaming replication.

```
version: '3.8'
```

```
services:
```

```
  postgres-primary:
```

```
    image: postgres:15
```

```
    container_name: postgres_primary
```

```
    environment:
```

```
      POSTGRES_USER: blockchain
```

```
      POSTGRES_PASSWORD: ${DB_PASSWORD}
```

```
      POSTGRES_DB: blockchain_index
```

```
    volumes:
```

```
      - postgres_primary_data:/var/lib/postgresql/data
```

```
      - ./init-primary.sh:/docker-entrypoint-initdb.d/init.sh
```

```
    ports:
```

```
      - "5432:5432"
```

```
    command: >
```

```
  postgres
```

```
    -c wal_level=replica
```

```
    -c max_wal_senders=3
```

-c max_replication_slots = 3
-c hot_standby = on
healthcheck:
test: ["CMD-SHELL", "pg_isready -U blockchain"]
interval: 10s
timeout: 5s
retries: 5

postgres-replica:
image: postgres:15
container_name: postgres_replica
environment:
POSTGRES_USER: blockchain
POSTGRES_PASSWORD: \${DB_PASSWORD}
PGUSER: replicator
PGPASSWORD: \${REPLICATION_PASSWORD}
volumes:
- postgres_replica_data:/var/lib/postgresql/data
- ./init-replica.sh:/docker-entrypoint-initdb.d/init.sh
ports:
- "5433:5432"
depends_on:
postgres-primary:
condition: service_healthy
command: >
postgres
-c hot_standby = on

pgbouncer:
image: bitnami/pgbouncer:latest
container_name: pgbouncer
environment:
POSTGRES_HOST: postgres-primary
POSTGRES_PORT: 5432
POSTGRES_USERNAME: blockchain
POSTGRES_PASSWORD: \${DB_PASSWORD}
POSTGRES_DATABASE: blockchain_index
PGBOUNCER_POOL_MODE: transaction
PGBOUNCER_MAX_CLIENT_CONN: 1000
PGBOUNCER_DEFAULT_POOL_SIZE: 50

ports:

- "6432:6432"

depends_on:

- postgres-primary
- postgres-replica

volumes:

postgres_primary_data:

postgres_replica_data:

The primary initialization script creates the replication user.

```
#!/bin/bash
```

```
psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --  
dbname "$POSTGRES_DB" <<-EOSQL  
CREATE USER replicator WITH REPLICATION ENCRYPTED  
PASSWORD '${REPLICATION_PASSWORD}';  
SELECT pg_create_physical_replication_slot('replica_slot');  
EOSQL
```

```
echo "host replication replicator 0.0.0.0/0 md5" >> /var/lib/  
postgresql/data/pg_hba.conf
```

The replica initialization script sets up streaming replication.

```
#!/bin/bash
```

```
if [ ! -s "/var/lib/postgresql/data/PG_VERSION" ]; then  
rm -rf /var/lib/postgresql/data/*
```

```
until pg_basebackup -h postgres-primary -D /var/lib/postgresql/  
data -U replicator -v -P -W  
do  
echo "Waiting for primary to be ready..."  
sleep 5  
done
```

```
cat > /var/lib/postgresql/data/postgresql.auto.conf <<EOF  
primary_conninfo = 'host=postgres-primary port=5432
```

```
user = replicator password = ${REPLICATION_PASSWORD}'
primary_slot_name = 'replica_slot'
EOF
```

```
touch /var/lib/postgresql/data/standby.signal
fi
```

The application connection manager routes reads to replicas and writes to the primary.

```
import asyncpg
from typing import Optional, List
import random

class ReplicatedDatabasePool:
    def __init__(
        self,
        primary_url: str,
        replica_urls: List[str],
        min_size: int = 5,
        max_size: int = 20
    ):
        self.primary_url = primary_url
        self.replica_urls = replica_urls
        self.min_size = min_size
        self.max_size = max_size
        self.primary_pool: Optional[asyncpg.Pool] = None
        self.replica_pools: List[asyncpg.Pool] = []
        self.replica_healthy: List[bool] = []

    async def connect(self):
        self.primary_pool = await asyncpg.create_pool(
            self.primary_url,
            min_size=self.min_size,
            max_size=self.max_size
        )

        for url in self.replica_urls:
            try:
                pool = await asyncpg.create_pool(
```



```

url,
min_size = self.min_size,
max_size = self.max_size
)
self.replica_pools.append(pool)
self.replica_healthy.append(True)
except Exception as e:
print(f"Failed to connect to replica {url}: {e}")

def _get_read_pool(self) -> asyncpg.Pool:
healthy_replicas = [
(i, pool) for i, pool in enumerate(self.replica_pools)
if self.replica_healthy[i]
]

if healthy_replicas:
_, pool = random.choice(healthy_replicas)
return pool

return self.primary_pool

async def execute(self, query: str, *args) -> str:
return await self.primary_pool.execute(query, *args)

async def fetch(self, query: str, *args) -> List:
pool = self._get_read_pool()
try:
return await pool.fetch(query, *args)
except Exception as e:
for i, p in enumerate(self.replica_pools):
if p == pool:
self.replica_healthy[i] = False
return await self.primary_pool.fetch(query, *args)

async def fetchrow(self, query: str, *args):
pool = self._get_read_pool()
try:
return await pool.fetchrow(query, *args)
except Exception as e:
for i, p in enumerate(self.replica_pools):

```

```
if p == pool:
    self.replica_healthy[i] = False
    return await self.primary_pool.fetchrow(query, *args)
```

```
async def fetchval(self, query: str, *args):
    pool = self._get_read_pool()
    try:
        return await pool.fetchval(query, *args)
    except Exception as e:
        for i, p in enumerate(self.replica_pools):
            if p == pool:
                self.replica_healthy[i] = False
        return await self.primary_pool.fetchval(query, *args)
```

```
async def check_replica_health(self):
    for i, pool in enumerate(self.replica_pools):
        try:
            await pool.fetchval("SELECT 1")
            self.replica_healthy[i] = True
        except Exception:
            self.replica_healthy[i] = False
```

```
async def close(self):
    if self.primary_pool:
        await self.primary_pool.close()
    for pool in self.replica_pools:
        await pool.close()
```

Reads are distributed across replicas. Writes go to the primary. Failed replicas are marked unhealthy and traffic routes around them.

SECTION 4: REDIS SENTINEL FOR CACHE FAILOVER

Redis Sentinel provides automatic failover for Redis.

version: '3.8'

services:

redis-master:
image: redis:7-alpine
container_name: redis_master
command: redis-server --appendonly yes
volumes:
- redis_master_data:/data
ports:
- "6379:6379"

redis-replica-1:
image: redis:7-alpine
container_name: redis_replica_1
command: redis-server --replicaof redis-master 6379 --appendonly
yes
volumes:
- redis_replica_1_data:/data
depends_on:
- redis-master

redis-replica-2:
image: redis:7-alpine
container_name: redis_replica_2
command: redis-server --replicaof redis-master 6379 --appendonly
yes
volumes:
- redis_replica_2_data:/data
depends_on:
- redis-master

sentinel-1:
image: redis:7-alpine
container_name: sentinel_1
command: redis-sentinel /etc/redis/sentinel.conf
volumes:
- ./sentinel.conf:/etc/redis/sentinel.conf
depends_on:
- redis-master

sentinel-2:
image: redis:7-alpine

```
container_name: sentinel_2
command: redis-sentinel /etc/redis/sentinel.conf
volumes:
- ./sentinel.conf:/etc/redis/sentinel.conf
depends_on:
- redis-master
```

```
sentinel-3:
image: redis:7-alpine
container_name: sentinel_3
command: redis-sentinel /etc/redis/sentinel.conf
volumes:
- ./sentinel.conf:/etc/redis/sentinel.conf
depends_on:
- redis-master
```

```
volumes:
redis_master_data:
redis_replica_1_data:
redis_replica_2_data:
```

The Sentinel configuration file.

```
sentinel monitor mymaster redis-master 6379 2
sentinel down-after-milliseconds mymaster 5000
sentinel failover-timeout mymaster 60000
sentinel parallel-syncs mymaster 1
```

The Python client for Sentinel-managed Redis.

```
from redis.asyncio import Sentinel
from typing import Optional, Any
import json
```

```
class SentinelRedisClient:
    def __init__(
        self,
        sentinel_hosts: list,
        master_name: str = "mymaster",
        password: str = None
    ):
```

```

):
self.sentinel = Sentinel(
sentinel_hosts,
socket_timeout=0.5
)
self.master_name = master_name
self.password = password
self.master = None
self.slave = None

async def connect(self):
self.master = self.sentinel.master_for(
self.master_name,
password=self.password
)
self.slave = self.sentinel.slave_for(
self.master_name,
password=self.password
)

async def get(self, key: str) -> Optional[Any]:
try:
value = await self.slave.get(key)
if value:
return json.loads(value)
return None
except Exception:
value = await self.master.get(key)
if value:
return json.loads(value)
return None

async def set(
self,
key: str,
value: Any,
ttl_seconds: int = None
):
serialized = json.dumps(value)
if ttl_seconds:

```

```

await self.master.setex(key, ttl_seconds, serialized)
else:
await self.master.set(key, serialized)

async def delete(self, key: str):
await self.master.delete(key)

async def get_master_info(self) -> dict:
return await self.sentinel.discover_master(self.master_name)

async def get_slaves_info(self) -> list:
return await self.sentinel.discover_slaves(self.master_name)

async def close(self):
if self.master:
await self.master.close()
if self.slave:
await self.slave.close()

```

Sentinel monitors Redis and promotes a replica to master when the primary fails. The client automatically discovers the current master.

SECTION 5: LOAD BALANCING

Distribute traffic across multiple service instances.

```

from typing import List, Dict, Optional
from dataclasses import dataclass, field
import random
import time
import asyncio
import aiohttp

```

```

@dataclass
class Backend:
    url: str
    weight: int = 1
    healthy: bool = True
    consecutive_failures: int = 0

```

```
last_check: float = 0
active_connections: int = 0
```

```
class LoadBalancer:
    def __init__(
        self,
        backends: List[Dict],
        health_check_interval: int = 30,
        failure_threshold: int = 3
    ):
        self.backends = [
            Backend(url=b["url"], weight=b.get("weight", 1))
            for b in backends
        ]
        self.health_check_interval = health_check_interval
        self.failure_threshold = failure_threshold
        self.round_robin_index = 0

    def get_healthy_backends(self) -> List[Backend]:
        return [b for b in self.backends if b.healthy]

    def select_round_robin(self) -> Optional[Backend]:
        healthy = self.get_healthy_backends()
        if not healthy:
            return None

        self.round_robin_index = (self.round_robin_index + 1) %
            len(healthy)
        return healthy[self.round_robin_index]

    def select_weighted_random(self) -> Optional[Backend]:
        healthy = self.get_healthy_backends()
        if not healthy:
            return None

        total_weight = sum(b.weight for b in healthy)
        r = random.uniform(0, total_weight)

        current = 0
        for backend in healthy:
```

```
current += backend.weight
if r <= current:
    return backend
```

```
return healthy[-1]
```

```
def select_least_connections(self) -> Optional[Backend]:
    healthy = self.get_healthy_backends()
    if not healthy:
        return None
```

```
return min(healthy, key=lambda b: b.active_connections)
```

```
async def forward_request(
    self,
    method: str,
    path: str,
    data: Dict = None,
    headers: Dict = None
) -> Dict:
```

```
backend = self.select_least_connections()
```

```
if not backend:
    return {"error": "No healthy backends"}
```

```
backend.active_connections += 1
```

```
try:
    url = backend.url.rstrip("/") + "/" + path.lstrip("/")
```

```
    async with aiohttp.ClientSession() as session:
        if method == "GET":
            async with session.get(url, headers=headers) as response:
                result = await response.json()
        elif method == "POST":
            async with session.post(url, json=data, headers=headers) as response:
                result = await response.json()
        else:
```



```

result = {"error": f"Unsupported method: {method}"}

backend.consecutive_failures = 0
return result

except Exception as e:
    backend.consecutive_failures += 1

if backend.consecutive_failures >= self.failure_threshold:
    backend.healthy = False

healthy = self.get_healthy_backends()
if healthy:
    return await self.forward_request(method, path, data, headers)

return {"error": str(e)}

finally:
    backend.active_connections -= 1

async def health_check_loop(self):
    while True:
        await self._check_all_backends()
        await asyncio.sleep(self.health_check_interval)

async def _check_all_backends(self):
    for backend in self.backends:
        try:
            async with aiohttp.ClientSession() as session:
                url = backend.url.rstrip("/") + "/health"
                async with session.get(url, timeout=5) as response:
                    if response.status == 200:
                        backend.healthy = True
                        backend.consecutive_failures = 0
                    else:
                        backend.consecutive_failures += 1
        except Exception:
            backend.consecutive_failures += 1

if backend.consecutive_failures >= self.failure_threshold:

```

```
backend.healthy = False
```

```
backend.last_check = time.time()
```

```
class ConsistentHashLoadBalancer:
```

```
    def __init__(self, backends: List[Dict], replicas: int = 100):  
        self.backends = {b["url"]: Backend(url=b["url"]) for b in  
        backends}
```

```
        self.replicas = replicas
```

```
        self.ring: Dict[int, str] = {}
```

```
        self._build_ring()
```

```
    def _hash(self, key: str) -> int:
```

```
        import hashlib
```

```
        return int(hashlib.md5(key.encode()).hexdigest(), 16)
```

```
    def _build_ring(self):
```

```
        self.ring = {}
```

```
        for url in self.backends:
```

```
            for i in range(self.replicas):
```

```
                key = f"{url}:{i}"
```

```
                hash_val = self._hash(key)
```

```
                self.ring[hash_val] = url
```

```
        self.sorted_keys = sorted(self.ring.keys())
```

```
    def get_backend(self, key: str) -> Optional[Backend]:
```

```
        if not self.sorted_keys:
```

```
            return None
```

```
        hash_val = self._hash(key)
```

```
        for ring_key in self.sorted_keys:
```

```
            if ring_key >= hash_val:
```

```
                url = self.ring[ring_key]
```

```
                backend = self.backends[url]
```

```
                if backend.healthy:
```

```
                    return backend
```

```

for ring_key in self.sorted_keys:
    url = self.ring[ring_key]
    backend = self.backends[url]
    if backend.healthy:
        return backend

return None

def add_backend(self, url: str):
    self.backends[url] = Backend(url=url)
    self._build_ring()

def remove_backend(self, url: str):
    if url in self.backends:
        del self.backends[url]
    self._build_ring()

```

Load balancing distributes requests across instances. Consistent hashing keeps related requests on the same backend for cache locality.

SECTION 6: CIRCUIT BREAKER PATTERN

Prevent cascading failures by stopping requests to failing services.

```

from enum import Enum
from typing import Callable, Any
import asyncio
import time

class CircuitState(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitBreaker:
    def __init__(
        self,
        name: str,

```

```

failure_threshold: int = 5,
recovery_timeout: int = 30,
half_open_requests: int = 3
):
    self.name = name
    self.failure_threshold = failure_threshold
    self.recovery_timeout = recovery_timeout
    self.half_open_requests = half_open_requests

    self.state = CircuitState.CLOSED
    self.failure_count = 0
    self.success_count = 0
    self.last_failure_time = 0
    self.half_open_successes = 0

    def _should_attempt(self) -> bool:
        if self.state == CircuitState.CLOSED:
            return True

        if self.state == CircuitState.OPEN:
            if time.time() - self.last_failure_time >= self.recovery_timeout:
                self.state = CircuitState.HALF_OPEN
                self.half_open_successes = 0
                return True
            return False

        if self.state == CircuitState.HALF_OPEN:
            return True

        return False

    def _record_success(self):
        self.failure_count = 0

        if self.state == CircuitState.HALF_OPEN:
            self.half_open_successes += 1
            if self.half_open_successes >= self.half_open_requests:
                self.state = CircuitState.CLOSED

        self.success_count += 1

```

```

def _record_failure(self):
    self.failure_count += 1
    self.last_failure_time = time.time()

    if self.state == CircuitState.HALF_OPEN:
        self.state = CircuitState.OPEN
    elif self.failure_count >= self.failure_threshold:
        self.state = CircuitState.OPEN

    async def execute(self, func: Callable, *args, **kwargs) -> Any:
        if not self._should_attempt():
            raise CircuitOpenError(f"Circuit {self.name} is open")

        try:
            if asyncio.iscoroutinefunction(func):
                result = await func(*args, **kwargs)
            else:
                result = func(*args, **kwargs)

        self._record_success()
        return result

    except Exception as e:
        self._record_failure()
        raise

    def get_state(self) -> Dict:
        return {
            "name": self.name,
            "state": self.state.value,
            "failure_count": self.failure_count,
            "success_count": self.success_count,
            "last_failure": self.last_failure_time
        }

class CircuitOpenError(Exception):
    pass

```

```

class CircuitBreakerRegistry:
    def __init__(self):
        self.breakers: Dict[str, CircuitBreaker] = {}

    def get_or_create(
        self,
        name: str,
        failure_threshold: int = 5,
        recovery_timeout: int = 30
    ) -> CircuitBreaker:

        if name not in self.breakers:
            self.breakers[name] = CircuitBreaker(
                name=name,
                failure_threshold=failure_threshold,
                recovery_timeout=recovery_timeout
            )

        return self.breakers[name]

    def get_all_states(self) -> Dict[str, Dict]:
        return {name: cb.get_state() for name, cb in self.breakers.items()}

def circuit_breaker(name: str, registry: CircuitBreakerRegistry):
    def decorator(func):
        async def wrapper(*args, **kwargs):
            breaker = registry.get_or_create(name)
            return await breaker.execute(func, *args, **kwargs)
        return wrapper
    return decorator

class RPCCircuitBreaker:
    def __init__(self, rpc_client, registry: CircuitBreakerRegistry):
        self.rpc = rpc_client
        self.registry = registry

    async def call(self, method: str, params: list = None) -> Any:

```

```
breaker = self.registry.get_or_create(f"rpc_{method}")
```

```
try:
```

```
    return await breaker.execute(self.rpc.call, method, params)
```

```
except CircuitOpenError:
```

```
    raise RPCUnavailableError(f"RPC method {method} circuit is  
open")
```

```
class RPCUnavailableError(Exception):
```

```
    pass
```

Circuit breakers prevent a failing service from consuming resources. After failures exceed the threshold, the circuit opens and requests fail fast.

SECTION 7: GRACEFUL DEGRADATION

When services fail, degrade gracefully instead of crashing.

```
from typing import Dict, Any, Optional, Callable
```

```
from dataclasses import dataclass
```

```
from enum import Enum
```

```
class ServiceLevel(Enum):
```

```
    FULL = "full"
```

```
    DEGRADED = "degraded"
```

```
    MINIMAL = "minimal"
```

```
    OFFLINE = "offline"
```

```
@dataclass
```

```
class FallbackConfig:
```

```
    primary: Callable
```

```
    fallback: Callable
```

```
    cache_key: Optional[str] = None
```

```
    stale_ttl: int = 3600
```

```
class GracefulDegradation:
```

```
    def __init__(self, cache):
```

```

self.cache = cache
self.service_levels: Dict[str, ServiceLevel] = {}
self.fallbacks: Dict[str, FallbackConfig] = {}

def set_service_level(self, service: str, level: ServiceLevel):
    self.service_levels[service] = level

def get_service_level(self, service: str) -> ServiceLevel:
    return self.service_levels.get(service, ServiceLevel.FULL)

def register_fallback(
    self,
    name: str,
    primary: Callable,
    fallback: Callable,
    cache_key: str = None
):
    self.fallbacks[name] = FallbackConfig(
        primary=primary,
        fallback=fallback,
        cache_key=cache_key
    )

    async def execute(self, name: str, *args, **kwargs) -> Any:
        if name not in self.fallbacks:
            raise ValueError(f"No fallback registered for {name}")

        config = self.fallbacks[name]

        try:
            if asyncio.iscoroutinefunction(config.primary):
                result = await config.primary(*args, **kwargs)
            else:
                result = config.primary(*args, **kwargs)

            if config.cache_key:
                cache_key = config.cache_key.format(*args, **kwargs)
                await self.cache.set(
                    f'fallback:{cache_key}',
                    result,

```



```
ttl_seconds = config.stale_ttl
)
```

```
return result
```

```
except Exception as primary_error:
if config.cache_key:
cache_key = config.cache_key.format(*args, **kwargs)
cached = await self.cache.get(f'fallback:{cache_key}')
if cached:
return {
"data": cached,
"stale": True,
"source": "cache"
}
```

```
try:
if asyncio.iscoroutinefunction(config.fallback):
result = await config.fallback(*args, **kwargs)
else:
result = config.fallback(*args, **kwargs)
```

```
return {
"data": result,
"degraded": True,
"source": "fallback"
}
```

```
except Exception as fallback_error:
return {
"error": str(primary_error),
"fallback_error": str(fallback_error),
"source": "none"
}
```

```
class DegradedBlockchainService:
def __init__(
self,
rpc_client,
```

```

database,
cache,
degradation: GracefulDegradation
):
self.rpc = rpc_client
self.db = database
self.cache = cache
self.degradation = degradation
self._setup_fallbacks()

```

```

def _setup_fallbacks(self):
self.degradation.register_fallback(
"get_balance",
primary=self._get_balance_rpc,
fallback=self._get_balance_db,
cache_key="balance:{0}"
)

```

```

self.degradation.register_fallback(
"get_token_price",
primary=self._get_price_api,
fallback=self._get_price_cached,
cache_key="price:{0}"
)

```

```

self.degradation.register_fallback(
"get_nft_metadata",
primary=self._get_metadata_ipfs,
fallback=self._get_metadata_db,
cache_key="metadata:{0}:{1}"
)

```

```

async def _get_balance_rpc(self, address: str) -> int:
result = await self.rpc.call("eth_getBalance", [address, "latest"])
return int(result, 16)

```

```

async def _get_balance_db(self, address: str) -> int:
row = await self.db.fetchrow(
"""

```

```

SELECT balance FROM account_balances

```

```

WHERE address = $1
ORDER BY block_number DESC
LIMIT 1
"""
    ,
    address.lower()
)
return row["balance"] if row else 0

```

```

async def _get_price_api(self, token: str) -> float:
    async with aiohttp.ClientSession() as session:
        url = f"https://api.coingecko.com/api/v3/simple/price?
ids={token}&vs_currencies=usd"
    async with session.get(url) as response:
        data = await response.json()
    return data[token]["usd"]

```

```

async def _get_price_cached(self, token: str) -> float:
    cached = await self.cache.get(f'price:{token}')
    if cached:
        return cached
    return 0.0

```

```

async def _get_metadata_ipfs(self, contract: str, token_id: str) ->
Dict:
    pass

```

```

async def _get_metadata_db(self, contract: str, token_id: str) ->
Dict:
    row = await self.db.fetchrow(
        """
        SELECT metadata FROM nft_metadata
        WHERE contract_address = $1 AND token_id = $2
        """
        ,
        contract.lower(),
        token_id
    )
    return row["metadata"] if row else {}

```

```

async def get_balance(self, address: str) -> Dict:
    return await self.degradation.execute("get_balance", address)

```

```
async def get_token_price(self, token: str) -> Dict:
    return await self.degradation.execute("get_token_price", token)
```

```
async def get_nft_metadata(self, contract: str, token_id: str) -> Dict:
    return await self.degradation.execute("get_nft_metadata", contract,
    token_id)
```

Graceful degradation returns stale cached data or simpler fallback results when primary services fail.

SECTION 8: KUBERNETES DEPLOYMENT

Container orchestration automates deployment, scaling, and failover.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: blockchain-api
  labels:
    app: blockchain-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: blockchain-api
  strategy:
    type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 0
  template:
    metadata:
      labels:
        app: blockchain-api
    spec:
      containers:
        - name: api
```

image: blockchain-api:latest
ports:
- containerPort: 8080
env:
- name: DATABASE_URL
valueFrom:
secretKeyRef:
name: blockchain-secrets
key: database-url
- name: REDIS_URL
valueFrom:
secretKeyRef:
name: blockchain-secrets
key: redis-url
- name: RPC_URL
valueFrom:
secretKeyRef:
name: blockchain-secrets
key: rpc-url
resources:
requests:
memory: "256Mi"
cpu: "250m"
limits:
memory: "512Mi"
cpu: "500m"
livenessProbe:
httpGet:
path: /livez
port: 8080
initialDelaySeconds: 10
periodSeconds: 15
timeoutSeconds: 5
failureThreshold: 3
readinessProbe:
httpGet:
path: /readyz
port: 8080
initialDelaySeconds: 5
periodSeconds: 10

timeoutSeconds: 5
failureThreshold: 3
affinity:
podAntiAffinity:
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 100
podAffinityTerm:
labelSelector:
matchLabels:
app: blockchain-api
topologyKey: kubernetes.io/hostname

apiVersion: v1
kind: Service
metadata:
name: blockchain-api
spec:
selector:
app: blockchain-api
ports:
- port: 80
targetPort: 8080
type: ClusterIP

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
name: blockchain-api-hpa
spec:
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: blockchain-api
minReplicas: 3
maxReplicas: 10
metrics:
- type: Resource
resource:

```
name: cpu
target:
type: Utilization
averageUtilization: 70
- type: Resource
resource:
name: memory
target:
type: Utilization
averageUtilization: 80
```

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: blockchain-api-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: blockchain-api
```

Kubernetes provides automated restarts, rolling updates, horizontal scaling, and pod anti-affinity to spread instances across nodes.

SECTION 9: DISASTER RECOVERY

Plan for catastrophic failures with backup and recovery procedures.

```
import subprocess
from datetime import datetime
from typing import Dict, List
import asyncio
import boto3
```

```
class DisasterRecovery:
    def __init__(self, config: Dict):
        self.config = config
        self.s3_client = boto3.client(
```

```

's3',
aws_access_key_id = config.get("aws_access_key"),
aws_secret_access_key = config.get("aws_secret_key"),
region_name = config.get("aws_region", "us-east-1")
)
self.bucket = config.get("backup_bucket")

async def backup_database(self) -> str:
timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
filename = f"blockchain_db_{timestamp}.sql.gz"
local_path = f"/tmp/{filename}"

db_url = self.config["database_url"]

dump_cmd = f'pg_dump {db_url} | gzip > {local_path}'
process = await asyncio.create_subprocess_shell(
dump_cmd,
stdout = asyncio.subprocess.PIPE,
stderr = asyncio.subprocess.PIPE
)
await process.communicate()

if process.returncode != 0:
raise Exception("Database backup failed")

s3_key = f"backups/database/{filename}"
self.s3_client.upload_file(local_path, self.bucket, s3_key)

subprocess.run(["rm", local_path])

return s3_key

async def restore_database(self, backup_key: str):
local_path = f"/tmp/"
restore_{datetime.utcnow().timestamp()}.sql.gz"

self.s3_client.download_file(self.bucket, backup_key, local_path)

db_url = self.config["database_url"]

```



```

restore_cmd = f'gunzip -c {local_path} | psql {db_url}'
process = await asyncio.create_subprocess_shell(
    restore_cmd,
    stdout=asyncio.subprocess.PIPE,
    stderr=asyncio.subprocess.PIPE
)
await process.communicate()

subprocess.run(["rm", local_path])

if process.returncode != 0:
    raise Exception("Database restore failed")

async def backup_redis(self) -> str:
    timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    filename = f"redis_dump_{timestamp}.rdb"

    redis_host = self.config["redis_host"]
    redis_port = self.config.get("redis_port", 6379)

    import redis
    r = redis.Redis(host=redis_host, port=redis_port)
    r.bgsave()

    await asyncio.sleep(5)

    local_path = f"/tmp/{filename}"
    subprocess.run(["cp", f"{redis_host}:/data/dump.rdb", local_path])

    s3_key = f"backups/redis/{filename}"
    self.s3_client.upload_file(local_path, self.bucket, s3_key)

    return s3_key

def list_backups(self, prefix: str = "backups/") -> List[Dict]:
    response = self.s3_client.list_objects_v2(
        Bucket=self.bucket,
        Prefix=prefix
    )

```

```
backups = []
for obj in response.get("Contents", []):
    backups.append({
        "key": obj["Key"],
        "size": obj["Size"],
        "modified": obj["LastModified"].isoformat()
    })

return sorted(backups, key=lambda x: x["modified"],
reverse=True)
```

```
async def run_backup_schedule(self):
    while True:
        try:
            db_backup = await self.backup_database()
            print(f"Database backup complete: {db_backup}")
```

```
            redis_backup = await self.backup_redis()
            print(f"Redis backup complete: {redis_backup}")
```

```
        await self._cleanup_old_backups()
```

```
    except Exception as e:
        print(f"Backup failed: {e}")
```

```
    await asyncio.sleep(3600)
```

```
async def _cleanup_old_backups(self, keep_days: int = 30):
    cutoff = datetime.utcnow().timestamp() - (keep_days * 86400)
```

```
    backups = self.list_backups()
```

```
    for backup in backups:
        modified =
            datetime.fromisoformat(backup["modified"].replace("Z", ""))
        if modified.timestamp() < cutoff:
            self.s3_client.delete_object(
                Bucket=self.bucket,
                Key=backup["key"]
            )
```

Regular backups to off-site storage enable recovery from catastrophic failures like data center outages or ransomware attacks.

SECTION 10: PUTTING IT ALL TOGETHER

Here is a complete high availability architecture.

```
class HighAvailabilityStack:
    def __init__(self, config: Dict):
        self.config = config
        self.circuit_registry = CircuitBreakerRegistry()

    async def initialize(self):
        self.db_pool = ReplicatedDatabasePool(
            primary_url=self.config["database_primary"],
            replica_urls=self.config["database_replicas"],
            min_size=5,
            max_size=20
        )
        await self.db_pool.connect()

        self.cache = SentinelRedisClient(
            sentinel_hosts=self.config["sentinel_hosts"],
            master_name="mymaster"
        )
        await self.cache.connect()

    from infrastructure.rpc import ResilientRPCClient
    self.rpc = ResilientRPCClient(self.config["rpc_providers"])

    self.rpc_breaker = RPCCircuitBreaker(self.rpc, self.circuit_registry)

    self.degradation = GracefulDegradation(self.cache)

    self.service = DegradedBlockchainService(
        rpc_client=self.rpc_breaker,
        database=self.db_pool,
```

```
cache = self.cache,  
degradation = self.degradation  
)
```

```
self.load_balancer = LoadBalancer(  
backends = self.config["api_backends"],  
health_check_interval = 30  
)
```

```
self.dr = DisasterRecovery(self.config)
```

```
asyncio.create_task(self.load_balancer.health_check_loop())  
asyncio.create_task(self.db_pool.check_replica_health_loop())  
asyncio.create_task(self.dr.run_backup_schedule())
```

```
async def get_balance(self, address: str) -> Dict:  
    return await self.service.get_balance(address)
```

```
async def get_health(self) -> Dict:  
    db_healthy = True  
    try:  
        await self.db_pool.fetchval("SELECT 1")  
    except Exception:  
        db_healthy = False
```

```
cache_healthy = True  
try:  
    await self.cache.master.ping()  
except Exception:  
    cache_healthy = False
```

```
circuits = self.circuit_registry.get_all_states()  
open_circuits = [  
    name for name, state in circuits.items()  
    if state["state"] == "open"  
]
```

```
return {  
    "database": "healthy" if db_healthy else "unhealthy",  
    "cache": "healthy" if cache_healthy else "unhealthy",  
}
```

```
"circuits": circuits,  
"open_circuits": open_circuits,  
"overall": "healthy" if db_healthy and cache_healthy and not  
open_circuits else "degraded"  
}
```

```
async def shutdown(self):  
    await self.db_pool.close()  
    await self.cache.close()
```

```
async def main():  
    config = {  
        "database_primary": "postgresql://user:pass@primary:5432/  
blockchain",  
        "database_replicas": [  
            "postgresql://user:pass@replica1:5432/blockchain",  
            "postgresql://user:pass@replica2:5432/blockchain"  
        ],  
        "sentinel_hosts": [  
            ("sentinel1", 26379),  
            ("sentinel2", 26379),  
            ("sentinel3", 26379)  
        ],  
        "rpc_providers": [  
            {"name": "alchemy", "url": "https://eth-mainnet.alchemyapi.io/v2/  
key", "priority": 0},  
            {"name": "infura", "url": "https://mainnet.infura.io/v3/key",  
            "priority": 1}  
        ],  
        "api_backends": [  
            {"url": "http://api1:8080", "weight": 1},  
            {"url": "http://api2:8080", "weight": 1},  
            {"url": "http://api3:8080", "weight": 1}  
        ],  
        "backup_bucket": "blockchain-backups",  
        "aws_region": "us-east-1"  
    }
```

```
stack = HighAvailabilityStack(config)
```

```
await stack.initialize()
```

```
health = await stack.get_health()  
print(f'System health: {health}')
```

```
balance = await  
stack.get_balance("0x742d35Cc6634C0532925a3b844Bc9e7595f3bC28")  
print(f'Balance: {balance}')
```

```
if __name__ == "__main__":  
    asyncio.run(main())
```

This chapter covered high availability patterns for blockchain applications. You learned stateless service design for easy scaling, database replication with read replicas, Redis Sentinel for cache failover, load balancing across multiple instances, circuit breakers to prevent cascading failures, graceful degradation when services fail, Kubernetes for container orchestration, and disaster recovery with backups. Together these patterns create systems that survive component failures and maintain service during incidents.

CHAPTER 7: SECURITY IN PRODUCTION

Blockchain applications handle real money. A security breach means actual financial losses, not just data exposure. Private keys must be protected with extreme care. APIs must resist abuse. Infrastructure must withstand attacks. This chapter covers security practices for production blockchain systems.

SECTION 1: THE SECURITY MINDSET

Security in blockchain is different from traditional applications. There is no password reset for stolen funds. Transactions are irreversible. Smart contract exploits can drain millions in seconds.

Defense in depth means multiple layers of protection. If one layer fails, others still protect you. Assume every component will be compromised eventually. Design systems to limit damage when breaches occur.

The principle of least privilege means every component gets only the permissions it needs. A read-only API service does not need private keys. An indexer does not need database write access to production tables.

Security is a process, not a feature. Regular audits, penetration testing, and security reviews are essential. Vulnerabilities are discovered constantly. Your security must evolve.

SECTION 2: PRIVATE KEY MANAGEMENT

Private keys are the most critical secrets. Whoever has the key controls the funds. Never store private keys in code, configuration files, or environment variables in plain text.

```
from abc import ABC, abstractmethod
from typing import Optional, Dict, Any
import os
import json
import hashlib
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import
PBKDF2HMAC
import base64

class KeyStore(ABC):
    @abstractmethod
    async def get_key(self, key_id: str) -> Optional[str]:
        pass

    @abstractmethod
    async def store_key(self, key_id: str, private_key: str) -> bool:
        pass

    @abstractmethod
    async def delete_key(self, key_id: str) -> bool:
        pass
```

```

class EncryptedFileKeyStore(KeyStore):
    def __init__(self, file_path: str, master_password: str):
        self.file_path = file_path
        self.cipher = self._create_cipher(master_password)
        self.keys: Dict[str, str] = {}
        self._load()

    def _create_cipher(self, password: str) -> Fernet:
        salt = b'blockchain_keystore_salt'
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=480000,
        )
        key = base64.urlsafe_b64encode(kdf.derive(password.encode()))
        return Fernet(key)

    def _load(self):
        if not os.path.exists(self.file_path):
            self.keys = {}
            return

        with open(self.file_path, 'rb') as f:
            encrypted = f.read()

        try:
            decrypted = self.cipher.decrypt(encrypted)
            self.keys = json.loads(decrypted.decode())
        except Exception:
            raise ValueError("Failed to decrypt keystore. Wrong password?")

    def _save(self):
        data = json.dumps(self.keys).encode()
        encrypted = self.cipher.encrypt(data)

        with open(self.file_path, 'wb') as f:
            f.write(encrypted)

```



```
os.chmod(self.file_path, 0o600)
```

```
async def get_key(self, key_id: str) -> Optional[str]:  
    return self.keys.get(key_id)
```

```
async def store_key(self, key_id: str, private_key: str) -> bool:  
    self.keys[key_id] = private_key  
    self._save()  
    return True
```

```
async def delete_key(self, key_id: str) -> bool:  
    if key_id in self.keys:  
        del self.keys[key_id]  
        self._save()  
    return True  
    return False
```

```
class AWS SecretsManagerKeyStore(KeyStore):  
    def __init__(self, region: str = "us-east-1", prefix: str =  
"blockchain"):  
        import boto3  
        self.client = boto3.client('secretsmanager', region_name=region)  
        self.prefix = prefix
```

```
    def _secret_name(self, key_id: str) -> str:  
        return f"{self.prefix}/keys/{key_id}"
```

```
    async def get_key(self, key_id: str) -> Optional[str]:  
        try:  
            response = self.client.get_secret_value(  
                SecretId=self._secret_name(key_id)  
            )  
            return response['SecretString']  
        except self.client.exceptions.ResourceNotFoundException:  
            return None  
        except Exception as e:  
            raise KeyStoreError(f"Failed to get key: {e}")
```

```
    async def store_key(self, key_id: str, private_key: str) -> bool:
```

```
secret_name = self._secret_name(key_id)
```

```
try:
    self.client.create_secret(
        Name = secret_name,
        SecretString = private_key,
        Tags = [
            {'Key': 'application', 'Value': 'blockchain'},
            {'Key': 'type', 'Value': 'private_key'}
        ]
    )
except self.client.exceptions.ResourceExistsException:
    self.client.put_secret_value(
        SecretId = secret_name,
        SecretString = private_key
    )
```

```
return True
```

```
async def delete_key(self, key_id: str) -> bool:
    try:
        self.client.delete_secret(
            SecretId = self._secret_name(key_id),
            ForceDeleteWithoutRecovery = False
        )
    return True
except Exception:
    return False
```

```
class HashiCorpVaultKeyStore(KeyStore):
    def __init__(self, vault_url: str, token: str, mount_path: str =
"secret"):
        import hvac
        self.client = hvac.Client(url = vault_url, token = token)
        self.mount_path = mount_path

    if not self.client.is_authenticated():
        raise ValueError("Vault authentication failed")
```

```

async def get_key(self, key_id: str) -> Optional[str]:
    try:
        response = self.client.secrets.kv.v2.read_secret_version(
            path = f"blockchain/keys/{key_id}",
            mount_point = self.mount_path
        )
        return response['data']['data']['private_key']
    except Exception:
        return None

```

```

async def store_key(self, key_id: str, private_key: str) -> bool:
    self.client.secrets.kv.v2.create_or_update_secret(
        path = f"blockchain/keys/{key_id}",
        secret = {'private_key': private_key},
        mount_point = self.mount_path
    )
    return True

```

```

async def delete_key(self, key_id: str) -> bool:
    try:
        self.client.secrets.kv.v2.delete_metadata_and_all_versions(
            path = f"blockchain/keys/{key_id}",
            mount_point = self.mount_path
        )
        return True
    except Exception:
        return False

```

```

class KeyStoreError(Exception):
    pass

```

Key stores encrypt private keys at rest and provide controlled access. Use AWS Secrets Manager or HashiCorp Vault for production systems.

SECTION 3: HARDWARE SECURITY MODULES

For high-value wallets, use hardware security modules (HSMs) that

never expose the private key.

```
from abc import ABC, abstractmethod
from typing import Dict, Optional
import asyncio
```

```
class HSMInterface(ABC):
    @abstractmethod
    async def sign_transaction(self, tx_hash: bytes, key_id: str) -> bytes:
        pass
```

```
    @abstractmethod
    async def get_public_key(self, key_id: str) -> str:
        pass
```

```
    @abstractmethod
    async def generate_key(self, key_id: str) -> str:
        pass
```

```
class AWSCloudHSMSigner(HSMInterface):
    def __init__(self, cluster_id: str, hsm_user: str, hsm_password: str):
        self.cluster_id = cluster_id
        self.user = hsm_user
        self.password = hsm_password
        self.session = None
```

```
    async def connect(self):
        pass
```

```
    async def sign_transaction(self, tx_hash: bytes, key_id: str) -> bytes:
        pass
```

```
    async def get_public_key(self, key_id: str) -> str:
        pass
```

```
    async def generate_key(self, key_id: str) -> str:
        pass
```

```

class LedgerHardwareWallet(HSMInterface):
    def __init__(self, derivation_path: str = "m/44'/60'/0'/0/0"):
        self.derivation_path = derivation_path

    async def sign_transaction(self, tx_hash: bytes, key_id: str) -> bytes:
        from ledgereth import sign_transaction
        signature = sign_transaction(tx_hash, self.derivation_path)
        return signature

    async def get_public_key(self, key_id: str) -> str:
        from ledgereth import get_accounts
        accounts = get_accounts()
        return accounts[0] if accounts else None

    async def generate_key(self, key_id: str) -> str:
        raise NotImplementedError("Ledger keys are derived from seed")

```

```

class MultiSigSigner:
    def __init__(
        self,
        signers: list,
        threshold: int,
        multisig_address: str
    ):
        self.signers = signers
        self.threshold = threshold
        self.multisig_address = multisig_address

```

```

    async def propose_transaction(
        self,
        to: str,
        value: int,
        data: bytes
    ) -> str:
        pass

```

```

    async def sign_transaction(self, tx_id: str, signer_index: int) -> bool:
        if signer_index >= len(self.signers):
            raise ValueError("Invalid signer index")

```

```
signer = self.signers[signer_index]
return True
```

```
async def execute_transaction(self, tx_id: str) -> str:
    pass
```

```
class GnosisSafeSigner:
    def __init__(
        self,
        safe_address: str,
        rpc_client,
        owner_key_store: KeyStore
    ):
        self.safe_address = safe_address
        self.rpc = rpc_client
        self.key_store = owner_key_store
```

```
    async def propose_transaction(
        self,
        to: str,
        value: int,
        data: str,
        operation: int = 0
    ) -> Dict:
```

```
        nonce = await self._get_nonce()
```

```
        tx_hash = await self._get_transaction_hash(
            to, value, data, operation, nonce
        )
```

```
        owner_key = await self.key_store.get_key("safe_owner")
        signature = self._sign_hash(tx_hash, owner_key)
```

```
    return {
        "safe_tx_hash": tx_hash,
        "to": to,
        "value": value,
```

```
"data": data,  
"operation": operation,  
"nonce": nonce,  
"signatures": [signature]  
}
```

```
async def add_signature(  
    self,  
    proposal: Dict,  
    owner_id: str  
    ) -> Dict:
```

```
    owner_key = await self.key_store.get_key(owner_id)  
    signature = self._sign_hash(proposal["safe_tx_hash"], owner_key)  
    proposal["signatures"].append(signature)
```

```
    return proposal
```

```
async def execute_transaction(self, proposal: Dict) -> str:  
    pass
```

```
async def _get_nonce(self) -> int:  
    pass
```

```
async def _get_transaction_hash(  
    self,  
    to: str,  
    value: int,  
    data: str,  
    operation: int,  
    nonce: int  
    ) -> str:  
    pass
```

```
def _sign_hash(self, hash_to_sign: str, private_key: str) -> str:  
    from eth_account import Account  
    signed = Account.sign_message(  
        {"messageHash": bytes.fromhex(hash_to_sign[2:])},  
        private_key  
    )
```

```
return signed.signature.hex()
```

HSMs and multisig wallets protect against single points of compromise. Even if one key is stolen, funds remain safe.

SECTION 4: API AUTHENTICATION AND AUTHORIZATION

Protect your APIs from unauthorized access.

```
import secrets
import hashlib
import hmac
import time
from typing import Optional, Dict, Set
from dataclasses import dataclass
from functools import wraps

@dataclass
class APIKey:
    key_id: str
    hashed_secret: str
    name: str
    permissions: Set[str]
    rate_limit: int
    created_at: float
    last_used: float
    active: bool

class APIKeyManager:
    def __init__(self, database):
        self.db = database
        self.key_cache: Dict[str, APIKey] = {}

    def generate_key(self) -> tuple:
        key_id = secrets.token_urlsafe(16)
        secret = secrets.token_urlsafe(32)
        hashed = hashlib.sha256(secret.encode()).hexdigest()

        return key_id, secret, hashed
```



```
async def create_key(
    self,
    name: str,
    permissions: Set[str],
    rate_limit: int = 100
) -> tuple:
```

```
key_id, secret, hashed = self.generate_key()
```

```
api_key = APIKey(
    key_id=key_id,
    hashed_secret=hashed,
    name=name,
    permissions=permissions,
    rate_limit=rate_limit,
    created_at=time.time(),
    last_used=0,
    active=True
)
```

```
await self.db.execute(
    """
```

```
INSERT INTO api_keys (key_id, hashed_secret, name, permissions,
rate_limit, created_at, active)
VALUES ($1, $2, $3, $4, $5, $6, $7)
""",
    key_id, hashed, name, list(permissions), rate_limit,
    api_key.created_at, True
)
```

```
full_key = f'{key_id}.{secret}'
return full_key, api_key
```

```
async def validate_key(self, full_key: str) -> Optional[APIKey]:
    try:
        key_id, secret = full_key.split('.', 1)
    except ValueError:
        return None
```

```

if key_id in self.key_cache:
    api_key = self.key_cache[key_id]
else:
    row = await self.db.fetchrow(
        "SELECT * FROM api_keys WHERE key_id = $1 AND active =
true",
        key_id
    )
    if not row:
        return None

    api_key = APIKey(
        key_id=row['key_id'],
        hashed_secret=row['hashed_secret'],
        name=row['name'],
        permissions=set(row['permissions']),
        rate_limit=row['rate_limit'],
        created_at=row['created_at'],
        last_used=row.get('last_used', 0),
        active=row['active']
    )
    self.key_cache[key_id] = api_key

    expected_hash = hashlib.sha256(secret.encode()).hexdigest()
    if not hmac.compare_digest(expected_hash, api_key.hashed_secret):
        return None

    await self.db.execute(
        "UPDATE api_keys SET last_used = $1 WHERE key_id = $2",
        time.time(), key_id
    )

    return api_key

async def revoke_key(self, key_id: str) -> bool:
    await self.db.execute(
        "UPDATE api_keys SET active = false WHERE key_id = $1",
        key_id
    )
    self.key_cache.pop(key_id, None)

```

```
return True
```

```
def has_permission(self, api_key: APIKey, permission: str) -> bool:  
    if "*" in api_key.permissions:  
        return True  
    return permission in api_key.permissions
```

```
class HMACAuthenticator:  
    def __init__(self, key_manager: APIKeyManager):  
        self.key_manager = key_manager
```

```
    def generate_signature(  
        self,  
        secret: str,  
        timestamp: str,  
        method: str,  
        path: str,  
        body: str = ""  
    ) -> str:
```

```
        message = f'{timestamp}{method}{path}{body}'  
        signature = hmac.new(  
            secret.encode(),  
            message.encode(),  
            hashlib.sha256  
        ).hexdigest()
```

```
        return signature
```

```
    async def verify_request(  
        self,  
        key_id: str,  
        signature: str,  
        timestamp: str,  
        method: str,  
        path: str,  
        body: str = ""  
    ) -> Optional[APIKey]:
```

```

request_time = int(timestamp)
current_time = int(time.time())

if abs(current_time - request_time) > 300:
    return None

row = await self.key_manager.db.fetchrow(
    "SELECT hashed_secret FROM api_keys WHERE key_id = $1 AND
    active = true",
    key_id
)

if not row:
    return None

return None

def require_permission(permission: str):
    def decorator(func):
        @wraps(func)
        async def wrapper(request, *args, **kwargs):
            api_key = request.get("api_key")

            if not api_key:
                return {"error": "Authentication required"}, 401

            if not request.get("key_manager").has_permission(api_key,
            permission):
                return {"error": "Permission denied"}, 403

            return await func(request, *args, **kwargs)
        return wrapper
    return decorator

```

API keys with permissions control who can access what. HMAC signatures prevent replay attacks.

SECTION 5: RATE LIMITING

Protect against abuse with rate limiting.

```
import time
from typing import Dict, Optional, Tuple
from dataclasses import dataclass
import asyncio
```

```
@dataclass
class RateLimitConfig:
    requests_per_second: int = 10
    requests_per_minute: int = 100
    requests_per_hour: int = 1000
    burst_size: int = 20
```

```
class TokenBucketRateLimiter:
    def __init__(
        self,
        rate: float,
        capacity: int
    ):
        self.rate = rate
        self.capacity = capacity
        self.tokens: Dict[str, float] = {}
        self.last_update: Dict[str, float] = {}
```

```
    def _refill(self, key: str) -> float:
        now = time.time()
        last = self.last_update.get(key, now)
        elapsed = now - last
```

```
        current_tokens = self.tokens.get(key, self.capacity)
        new_tokens = min(self.capacity, current_tokens + elapsed *
self.rate)
```

```
        self.tokens[key] = new_tokens
        self.last_update[key] = now
```

```
    return new_tokens
```

```

def check(self, key: str, cost: int = 1) -> Tuple[bool, float]:
    tokens = self._refill(key)

    if tokens >= cost:
        self.tokens[key] = tokens - cost
        return True, 0

    wait_time = (cost - tokens) / self.rate
    return False, wait_time

    async def wait(self, key: str, cost: int = 1) -> bool:
        allowed, wait_time = self.check(key, cost)

        if allowed:
            return True

        if wait_time > 60:
            return False

        await asyncio.sleep(wait_time)
        return self.check(key, cost)[0]

class RedisRateLimiter:
    def __init__(self, redis_client, prefix: str = "ratelimit"):
        self.redis = redis_client
        self.prefix = prefix

    async def check_rate_limit(
        self,
        key: str,
        limit: int,
        window_seconds: int
    ) -> Tuple[bool, int, int]:

        redis_key = f'{self.prefix}:{key}'
        now = time.time()
        window_start = now - window_seconds

        pipe = self.redis.pipeline()

```

```
pipe.zremrangebyscore(redis_key, 0, window_start)
pipe.zadd(redis_key, {str(now): now})
pipe.zcard(redis_key)
pipe.expire(redis_key, window_seconds)
```

```
results = await pipe.execute()
current_count = results[2]
```

```
if current_count > limit:
    await self.redis.zrem(redis_key, str(now))
    remaining = 0
    reset_time = int(window_start + window_seconds)
    return False, remaining, reset_time
```

```
remaining = limit - current_count
reset_time = int(now + window_seconds)
```

```
return True, remaining, reset_time
```

```
async def check_multi_window(
    self,
    key: str,
    limits: Dict[str, Tuple[int, int]]
) -> Tuple[bool, Dict]:
```

```
results = {}
```

```
for window_name, (limit, seconds) in limits.items():
    window_key = f'{key}:{window_name}'
    allowed, remaining, reset = await self.check_rate_limit(
        window_key, limit, seconds
    )
    results[window_name] = {
        "allowed": allowed,
        "remaining": remaining,
        "reset": reset
    }
```

```
if not allowed:
    return False, results
```

```
return True, results
```

```
class AdaptiveRateLimiter:
    def __init__(
        self,
        redis_limiter: RedisRateLimiter,
        base_limit: int = 100,
        min_limit: int = 10,
        max_limit: int = 500
    ):
        self.limiter = redis_limiter
        self.base_limit = base_limit
        self.min_limit = min_limit
        self.max_limit = max_limit
        self.reputation: Dict[str, float] = {}

    async def check(self, key: str, action: str = "request") ->
    Tuple[bool, Dict]:
        reputation = self.reputation.get(key, 1.0)
        effective_limit = int(self.base_limit * reputation)
        effective_limit = max(self.min_limit, min(self.max_limit,
        effective_limit))

        allowed, remaining, reset = await self.limiter.check_rate_limit(
            f'{key}:{action}',
            effective_limit,
            60
        )

        return allowed, {
            "limit": effective_limit,
            "remaining": remaining,
            "reset": reset,
            "reputation": reputation
        }

    def adjust_reputation(self, key: str, delta: float):
        current = self.reputation.get(key, 1.0)
```



```
new_reputation = max(0.1, min(2.0, current + delta))
self.reputation[key] = new_reputation
```

```
def penalize(self, key: str, severity: str = "minor"):
    penalties = {
        "minor": -0.1,
        "moderate": -0.3,
        "severe": -0.5
    }
    self.adjust_reputation(key, penalties.get(severity, -0.1))
```

```
def reward(self, key: str):
    self.adjust_reputation(key, 0.05)
```

Rate limiting prevents abuse and ensures fair resource allocation.

SECTION 6: INPUT VALIDATION AND SANITIZATION

Never trust user input. Validate and sanitize everything.

```
import re
from typing import Any, Optional, List, Dict
from dataclasses import dataclass
from enum import Enum
```

```
class ValidationError(Exception):
    def __init__(self, field: str, message: str):
        self.field = field
        self.message = message
        super().__init__(f'{field}: {message}')
```

```
class AddressValidator:
    ETHEREUM_ADDRESS_PATTERN = re.compile(r'^0x[a-fA-F0-9]{40}$')
```

```
@classmethod
def validate(cls, address: str) -> str:
    if not address:
        raise ValidationError("address", "Address is required")
```

```
if not cls.ETHEREUM_ADDRESS_PATTERN.match(address):
    raise ValidationError("address", "Invalid Ethereum address format")

return address.lower()
```

```
@classmethod
def validate_checksum(cls, address: str) -> str:
    from eth_utils import is_checksum_address, to_checksum_address
```

```
if not cls.ETHEREUM_ADDRESS_PATTERN.match(address):
    raise ValidationError("address", "Invalid address format")
```

```
if address != address.lower() and address != address.upper():
    if not is_checksum_address(address):
        raise ValidationError("address", "Invalid checksum")
```

```
return to_checksum_address(address)
```

```
class TransactionHashValidator:
```

```
    TX_HASH_PATTERN = re.compile(r'^0x[a-fA-F0-9]{64}$')
```

```
    @classmethod
```

```
    def validate(cls, tx_hash: str) -> str:
```

```
        if not tx_hash:
```

```
            raise ValidationError("tx_hash", "Transaction hash is required")
```

```
        if not cls.TX_HASH_PATTERN.match(tx_hash):
```

```
            raise ValidationError("tx_hash", "Invalid transaction hash format")
```

```
        return tx_hash.lower()
```

```
class HexDataValidator:
```

```
    @classmethod
```

```
    def validate(cls, data: str, max_length: int = None) -> str:
```

```
        if not data:
```

```
            return "0x"
```

```

if not data.startswith("0x"):
    raise ValidationError("data", "Hex data must start with 0x")

hex_part = data[2:]

if not re.match(r'^[a-fA-F0-9]*$', hex_part):
    raise ValidationError("data", "Invalid hex characters")

if len(hex_part) % 2 != 0:
    raise ValidationError("data", "Hex data must have even length")

if max_length and len(hex_part) > max_length * 2:
    raise ValidationError("data", f"Data exceeds maximum length of {max_length} bytes")

return data.lower()

```

```

class IntegerValidator:
    @classmethod
    def validate(
        cls,
        value: Any,
        field_name: str,
        min_value: int = None,
        max_value: int = None
    ) -> int:

        try:
            if isinstance(value, str):
                if value.startswith("0x"):
                    int_value = int(value, 16)
                else:
                    int_value = int(value)
            else:
                int_value = int(value)
        except (ValueError, TypeError):
            raise ValidationError(field_name, "Must be a valid integer")

        if min_value is not None and int_value < min_value:

```

```
raise ValidationError(field_name, f"Must be at least {min_value}")
```

```
if max_value is not None and int_value > max_value:
```

```
raise ValidationError(field_name, f"Must be at most {max_value}")
```

```
return int_value
```

```
class RequestValidator:
```

```
def __init__(self):
```

```
self.errors: List[ValidationError] = []
```

```
def validate_address(self, value: str, field: str = "address") ->  
Optional[str]:
```

```
try:
```

```
return AddressValidator.validate(value)
```

```
except ValidationError as e:
```

```
self.errors.append(e)
```

```
return None
```

```
def validate_tx_hash(self, value: str, field: str = "tx_hash") ->  
Optional[str]:
```

```
try:
```

```
return TransactionHashValidator.validate(value)
```

```
except ValidationError as e:
```

```
self.errors.append(e)
```

```
return None
```

```
def validate_integer(  
self,
```

```
value: Any,
```

```
field: str,
```

```
min_value: int = None,
```

```
max_value: int = None
```

```
) -> Optional[int]:
```

```
try:
```

```
return IntegerValidator.validate(value, field, min_value, max_value)
```

```
except ValidationError as e:
```

```
self.errors.append(e)
```

```
return None
```

```
def validate_required(self, value: Any, field: str) -> bool:
    if value is None or value == "":
        self.errors.append(ValidationError(field, "This field is required"))
    return False
    return True
```

```
def has_errors(self) -> bool:
    return len(self.errors) > 0
```

```
def get_errors(self) -> List[Dict]:
    return [{"field": e.field, "message": e.message} for e in self.errors]
```

```
def validate_api_request(schema: Dict):
```

```
    def decorator(func):
```

```
        @wraps(func)
```

```
        async def wrapper(request, *args, **kwargs):
```

```
            validator = RequestValidator()
```

```
            validated_data = {}
```

```
            for field, rules in schema.items():
```

```
                value = request.get(field)
```

```
                if rules.get("required") and not validator.validate_required(value,
field):
```

```
                    continue
```

```
                if value is None:
```

```
                    continue
```

```
                field_type = rules.get("type")
```

```
                if field_type == "address":
```

```
                    validated_data[field] = validator.validate_address(value, field)
```

```
                elif field_type == "tx_hash":
```

```
                    validated_data[field] = validator.validate_tx_hash(value, field)
```

```
                elif field_type == "integer":
```

```
                    validated_data[field] = validator.validate_integer(
value, field,
```

```
rules.get("min"),
rules.get("max")
)
```

```
if validator.has_errors():
    return {"errors": validator.get_errors()}, 400
```

```
request["validated_data"] = validated_data
return await func(request, *args, **kwargs)
return wrapper
return decorator
```

Validate all inputs before processing. Reject malformed data early.

SECTION 7: DDoS PROTECTION

Protect against distributed denial of service attacks.

```
import time
from typing import Dict, Set, Optional
from collections import defaultdict
import asyncio
```

```
class DDoSProtection:
    def __init__(
        self,
        redis_client,
        block_threshold: int = 100,
        block_duration: int = 3600
    ):
        self.redis = redis_client
        self.block_threshold = block_threshold
        self.block_duration = block_duration
        self.local_blocklist: Set[str] = set()
        self.request_counts: Dict[str, int] = defaultdict(int)
```

```
async def is_blocked(self, ip: str) -> bool:
    if ip in self.local_blocklist:
        return True
```

```
blocked = await self.redis.get(f"blocked:{ip}")
return blocked is not None
```

```
async def block_ip(self, ip: str, reason: str, duration: int = None):
    duration = duration or self.block_duration
```

```
    await self.redis.setex(
        f"blocked:{ip}",
        duration,
        reason
    )
```

```
    self.local_blocklist.add(ip)
```

```
    await self.redis.lpush(
        "security:blocked_ips",
        f"{time.time()}:{ip}:{reason}"
    )
```

```
async def track_request(self, ip: str) -> bool:
    key = f"ddos:requests:{ip}"
```

```
    pipe = self.redis.pipeline()
    pipe.incr(key)
    pipe.expire(key, 60)
    results = await pipe.execute()
```

```
    count = results[0]
```

```
    if count > self.block_threshold:
        await self.block_ip(ip, "Rate limit exceeded")
        return False
```

```
    return True
```

```
async def track_error(self, ip: str, error_type: str):
    key = f"ddos:errors:{ip}"
```

```
    await self.redis.hincrby(key, error_type, 1)
```

```
await self.redis.expire(key, 300)
```

```
total_errors = await self.redis.hvals(key)  
total = sum(int(v) for v in total_errors)
```

```
if total > 50:
```

```
await self.block_ip(ip, f"Too many errors: {error_type}")
```

```
class RequestThrottler:
```

```
def __init__(self, max_concurrent: int = 100):
```

```
self.semaphore = asyncio.Semaphore(max_concurrent)
```

```
self.pending_requests = 0
```

```
self.rejected_requests = 0
```

```
async def acquire(self, timeout: float = 5.0) -> bool:
```

```
try:
```

```
await asyncio.wait_for(
```

```
self.semaphore.acquire(),
```

```
timeout=timeout
```

```
)
```

```
self.pending_requests += 1
```

```
return True
```

```
except asyncio.TimeoutError:
```

```
self.rejected_requests += 1
```

```
return False
```

```
def release(self):
```

```
self.pending_requests -= 1
```

```
self.semaphore.release()
```

```
def get_stats(self) -> Dict:
```

```
return {
```

```
"pending": self.pending_requests,
```

```
"rejected": self.rejected_requests,
```

```
"available": self.semaphore._value
```

```
}
```

```
class GeoIPBlocker:
```



```
def __init__(self, blocked_countries: Set[str] = None):
    self.blocked_countries = blocked_countries or set()
```

```
async def check_ip(self, ip: str) -> Optional[str]:
    pass
```

```
async def is_allowed(self, ip: str) -> bool:
    country = await self.check_ip(ip)
    if country and country in self.blocked_countries:
        return False
    return True
```

```
class ChallengeSystem:
    def __init__(self, redis_client, difficulty: int = 4):
        self.redis = redis_client
        self.difficulty = difficulty
        self.challenge_prefix = "0" * difficulty
```

```
    def generate_challenge(self) -> Dict:
        import secrets
        challenge = secrets.token_hex(32)
        return {
            "challenge": challenge,
            "difficulty": self.difficulty
        }
```

```
    def verify_solution(self, challenge: str, solution: str) -> bool:
        combined = f'{challenge}{solution}'
        hash_result = hashlib.sha256(combined.encode()).hexdigest()
        return hash_result.startswith(self.challenge_prefix)
```

```
    async def require_challenge(self, ip: str) -> bool:
        suspicious_score = await self.redis.get(f'suspicious:{ip}')
        return suspicious_score and int(suspicious_score) > 5
```

DDoS protection combines rate limiting, IP blocking, and proof-of-work challenges to mitigate attacks.

SECTION 8: AUDIT LOGGING

Log all security-relevant events for investigation and compliance.

```
import json
import time
from typing import Dict, Any, Optional
from enum import Enum
from dataclasses import dataclass, asdict

class AuditEventType(Enum):
    AUTHENTICATION = "authentication"
    AUTHORIZATION = "authorization"
    KEY_ACCESS = "key_access"
    TRANSACTION_SIGN = "transaction_sign"
    TRANSACTION_SEND = "transaction_send"
    CONFIG_CHANGE = "config_change"
    API_KEY_CREATE = "api_key_create"
    API_KEY_REVOKE = "api_key_revoke"
    SECURITY_ALERT = "security_alert"
    DATA_ACCESS = "data_access"

@dataclass
class AuditEvent:
    timestamp: float
    event_type: AuditEventType
    actor: str
    action: str
    resource: str
    outcome: str
    ip_address: Optional[str]
    user_agent: Optional[str]
    details: Dict[str, Any]
    request_id: Optional[str]

class AuditLogger:
    def __init__(self, database, elasticsearch_sink = None):
        self.db = database
        self.es = elasticsearch_sink
        self.buffer: list = []
```

```
self.buffer_size = 100
```

```
async def log(
    self,
    event_type: AuditEventType,
    actor: str,
    action: str,
    resource: str,
    outcome: str,
    ip_address: str = None,
    user_agent: str = None,
    details: Dict = None,
    request_id: str = None
):
    event = AuditEvent(
        timestamp=time.time(),
        event_type=event_type,
        actor=actor,
        action=action,
        resource=resource,
        outcome=outcome,
        ip_address=ip_address,
        user_agent=user_agent,
        details=details or {},
        request_id=request_id
    )
```

```
self.buffer.append(event)
```

```
if len(self.buffer) >= self.buffer_size:
    await self.flush()
```

```
async def flush(self):
    if not self.buffer:
        return
```

```
events = self.buffer
self.buffer = []
```

```
records = [
```

```
(
e.timestamp,
e.event_type.value,
e.actor,
e.action,
e.resource,
e.outcome,
e.ip_address,
e.user_agent,
json.dumps(e.details),
e.request_id
)
for e in events
]
```

```
await self.db.executemany(
"""
```

```
INSERT INTO audit_log (timestamp, event_type, actor, action,
resource, outcome, ip_address, user_agent, details, request_id)
VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10)
""",
records
)
```

```
if self.es:
for event in events:
await self.es.log(asdict(event))
```

```
async def log_authentication(
self,
actor: str,
success: bool,
ip_address: str,
method: str,
details: Dict = None
):
await self.log(
event_type = AuditEventType.AUTHENTICATION,
actor = actor,
action = f'authenticate_{method}',
```

```
resource = "api",  
outcome = "success" if success else "failure",  
ip_address = ip_address,  
details = details  
)
```

```
async def log_key_access(  
self,  
actor: str,  
key_id: str,  
action: str,  
ip_address: str  
):  
await self.log(  
event_type = AuditEventType.KEY_ACCESS,  
actor = actor,  
action = action,  
resource = f"key:{key_id}",  
outcome = "success",  
ip_address = ip_address,  
details = {"key_id": key_id}  
)
```

```
async def log_transaction(  
self,  
actor: str,  
tx_hash: str,  
action: str,  
from_address: str,  
to_address: str,  
value: int,  
ip_address: str  
):  
await self.log(  
event_type = AuditEventType.TRANSACTION_SEND,  
actor = actor,  
action = action,  
resource = f"tx:{tx_hash}",  
outcome = "success",  
ip_address = ip_address,
```

```
details = {  
    "tx_hash": tx_hash,  
    "from": from_address,  
    "to": to_address,  
    "value": value  
}  
)
```

```
async def log_security_alert(  
    self,  
    alert_type: str,  
    severity: str,  
    source_ip: str,  
    details: Dict  
):  
    await self.log(  
        event_type = AuditEventType.SECURITY_ALERT,  
        actor = "system",  
        action = alert_type,  
        resource = "security",  
        outcome = severity,  
        ip_address = source_ip,  
        details = details  
    )
```

```
async def search(  
    self,  
    event_type: AuditEventType = None,  
    actor: str = None,  
    start_time: float = None,  
    end_time: float = None,  
    limit: int = 100  
    ) -> list:
```

```
    conditions = []  
    params = []  
    param_count = 0
```

```
    if event_type:  
        param_count += 1
```

```
conditions.append(f'event_type = ${param_count}')
params.append(event_type.value)
```

```
if actor:
    param_count += 1
    conditions.append(f'actor = ${param_count}')
    params.append(actor)
```

```
if start_time:
    param_count += 1
    conditions.append(f'timestamp >= ${param_count}')
    params.append(start_time)
```

```
if end_time:
    param_count += 1
    conditions.append(f'timestamp <= ${param_count}')
    params.append(end_time)
```

```
where_clause = " AND ".join(conditions) if conditions else "1 = 1"
```

```
param_count += 1
params.append(limit)
```

```
query = f"""
SELECT * FROM audit_log
WHERE {where_clause}
ORDER BY timestamp DESC
LIMIT ${param_count}
"""
```

```
rows = await self.db.fetch(query, *params)
return [dict(row) for row in rows]
```

Audit logs provide a trail of all security-relevant actions for investigation and compliance.

SECTION 9: SECURITY MONITORING

Detect and respond to security threats in real-time.

```
from typing import List, Dict, Callable
import asyncio
from collections import defaultdict
```

```
class SecurityMonitor:
    def __init__(
        self,
        audit_logger: AuditLogger,
        alert_manager,
        redis_client
    ):
        self.audit = audit_logger
        self.alerts = alert_manager
        self.redis = redis_client
        self.rules: List[SecurityRule] = []
        self.running = False
```

```
def add_rule(self, rule: 'SecurityRule'):
    self.rules.append(rule)
```

```
async def start(self):
    self.running = True
```

```
self._setup_default_rules()
```

```
while self.running:
    await self._check_all_rules()
    await asyncio.sleep(60)
```

```
def _setup_default_rules(self):
    self.add_rule(FailedAuthenticationRule(threshold = 10,
    window = 300))
    self.add_rule(UnusualTransactionRule(threshold = 100))
    self.add_rule(APIKeyAbuseRule(threshold = 1000, window = 3600))
    self.add_rule(NewIPForKeyRule())
    self.add_rule(LargeTransactionRule(threshold_eth = 10))
```

```
async def _check_all_rules(self):
    for rule in self.rules:
```



```

try:
    alerts = await rule.check(self.redis, self.audit)
    for alert in alerts:
        await self._handle_alert(alert)
except Exception as e:
    print(f"Security rule error: {e}")

```

```

async def _handle_alert(self, alert: Dict):
    await self.audit.log_security_alert(
        alert_type = alert["type"],
        severity = alert["severity"],
        source_ip = alert.get("ip"),
        details = alert
    )

```

```

await self.alerts._fire_alert(alert)

```

```

class SecurityRule:
    async def check(self, redis, audit) -> List[Dict]:
        raise NotImplementedError

```

```

class FailedAuthenticationRule(SecurityRule):
    def __init__(self, threshold: int, window: int):
        self.threshold = threshold
        self.window = window

```

```

    async def check(self, redis, audit) -> List[Dict]:
        alerts = []

```

```

        start_time = time.time() - self.window

```

```

        failed_auths = await audit.search(
            event_type=AuditEventType.AUTHENTICATION,
            start_time=start_time
        )

```

```

        ip_failures = defaultdict(int)
        for event in failed_auths:

```

```
if event["outcome"] == "failure":
    ip_failures[event["ip_address"]] += 1
```

```
for ip, count in ip_failures.items():
    if count >= self.threshold:
        alerts.append({
            "type": "brute_force_attempt",
            "severity": "high",
            "ip": ip,
            "failed_attempts": count,
            "window_seconds": self.window
        })
```

```
return alerts
```

```
class LargeTransactionRule(SecurityRule):
    def __init__(self, threshold_eth: float):
        self.threshold_wei = int(threshold_eth * 1e18)
```

```
    async def check(self, redis, audit) -> List[Dict]:
        alerts = []
```

```
        start_time = time.time() - 3600
```

```
        transactions = await audit.search(
            event_type=AuditEventType.TRANSACTION_SEND,
            start_time=start_time
        )
```

```
        for tx in transactions:
            details = json.loads(tx["details"]) if isinstance(tx["details"], str)
            else tx["details"]
            value = details.get("value", 0)
```

```
        if value >= self.threshold_wei:
            alerts.append({
                "type": "large_transaction",
                "severity": "medium",
                "tx_hash": details.get("tx_hash"),
```

```
"value_eth": value / 1e18,  
"from": details.get("from"),  
"to": details.get("to"),  
"actor": tx["actor"],  
"ip": tx["ip_address"]  
})
```

```
return alerts
```

```
class UnusualTransactionRule(SecurityRule):  
    def __init__(self, threshold: int):  
        self.threshold = threshold  
  
    async def check(self, redis, audit) -> List[Dict]:  
        alerts = []  
  
    return alerts
```

```
class APIKeyAbuseRule(SecurityRule):  
    def __init__(self, threshold: int, window: int):  
        self.threshold = threshold  
        self.window = window  
  
    async def check(self, redis, audit) -> List[Dict]:  
        return []
```

```
class NewIPForKeyRule(SecurityRule):  
    async def check(self, redis, audit) -> List[Dict]:  
        return []
```

Security monitoring detects anomalies and potential attacks in real-time.

SECTION 10: SECURITY CHECKLIST

Before deploying to production, verify these security requirements.

```

class SecurityChecklist:
    def __init__(self):
        self.checks = []

    def add_check(self, name: str, check_func: Callable, critical: bool =
False):
        self.checks.append({
            "name": name,
            "check": check_func,
            "critical": critical
        })

    async def run_all(self) -> Dict:
        results = []
        passed = 0
        failed = 0
        critical_failed = 0

        for check in self.checks:
            try:
                result = await check["check"]()
                status = "passed" if result else "failed"

                if not result:
                    failed += 1
                    if check["critical"]:
                        critical_failed += 1
                else:
                    passed += 1

            except Exception as e:
                status = "error"
                result = str(e)
                failed += 1
                if check["critical"]:
                    critical_failed += 1

        results.append({
            "name": check["name"],

```

```
"status": status,  
"critical": check["critical"],  
"details": result  
})
```

```
return {  
    "passed": passed,  
    "failed": failed,  
    "critical_failed": critical_failed,  
    "deployment_ready": critical_failed == 0,  
    "checks": results  
}
```

```
async def create_security_checklist(config: Dict) ->  
SecurityChecklist:  
    checklist = SecurityChecklist()
```

```
async def check_no_plaintext_keys():  
    import os  
    for key in ["PRIVATE_KEY", "SECRET_KEY", "API_SECRET"]:  
        if key in os.environ:  
            return False  
    return True
```

```
checklist.add_check(  
    "No plaintext keys in environment",  
    check_no_plaintext_keys,  
    critical = True  
)
```

```
async def check_tls_enabled():  
    return config.get("tls_enabled", False)
```

```
checklist.add_check(  
    "TLS enabled for all endpoints",  
    check_tls_enabled,  
    critical = True  
)
```

```
async def check_rate_limiting():  
    return config.get("rate_limiting_enabled", False)
```

```
checklist.add_check(  
    "Rate limiting configured",  
    check_rate_limiting,  
    critical = True  
)
```

```
async def check_audit_logging():  
    return config.get("audit_logging_enabled", False)
```

```
checklist.add_check(  
    "Audit logging enabled",  
    check_audit_logging,  
    critical = True  
)
```

```
async def check_key_encryption():  
    return config.get("key_store_type") in ["vault", "aws_secrets",  
    "encrypted_file"]
```

```
checklist.add_check(  
    "Key store uses encryption",  
    check_key_encryption,  
    critical = True  
)
```

```
async def check_input_validation():  
    return config.get("input_validation_enabled", False)
```

```
checklist.add_check(  
    "Input validation enabled",  
    check_input_validation,  
    critical = False  
)
```

```
async def check_ddos_protection():  
    return config.get("ddos_protection_enabled", False)
```

```

checklist.add_check(
    "DDoS protection configured",
    check_ddos_protection,
    critical = False
)

```

```

return checklist

```

```

async def main():
    config = {
        "tls_enabled": True,
        "rate_limiting_enabled": True,
        "audit_logging_enabled": True,
        "key_store_type": "vault",
        "input_validation_enabled": True,
        "ddos_protection_enabled": True
    }

```

```

checklist = await create_security_checklist(config)
results = await checklist.run_all()

```

```

print(f"Security Check Results:")
print(f"Passed: {results['passed']}")
print(f"Failed: {results['failed']}")
print(f"Deployment Ready: {results['deployment_ready']}")

```

```

for check in results["checks"]:
    status = "PASS" if check["status"] == "passed" else "FAIL"
    critical = " (CRITICAL)" if check["critical"] else ""
    print(f" [{status}] {check['name']}{critical}")

```

```

if __name__ == "__main__":
    asyncio.run(main())

```

This chapter covered security practices for production blockchain applications. You learned key management with encryption and HSMs, API authentication with keys and HMAC, rate limiting to prevent abuse, input validation to reject malformed data, DDoS protection with blocking and challenges, audit logging for

compliance and investigation, and security monitoring for threat detection. Security is not optional in blockchain. It is the difference between a successful application and a catastrophic loss.

CHAPTER 8: COST OPTIMIZATION

Blockchain infrastructure is expensive. RPC providers charge per request. Archive nodes require terabytes of storage. Indexing full chains demands significant compute. Without optimization, costs can exceed revenue. This chapter teaches strategies to minimize infrastructure spending while maintaining performance.

SECTION 1: UNDERSTANDING BLOCKCHAIN INFRASTRUCTURE COSTS

Blockchain applications have unique cost drivers that differ from traditional web applications.

RPC costs scale with request volume. Most providers charge per compute unit, where different methods cost different amounts. A simple `eth_blockNumber` costs less than a complex `eth_call`. Archive queries cost more than current state queries.

Storage costs grow continuously. Blockchains never shrink. Ethereum adds gigabytes daily. Indexing creates even more data. Without pruning strategies, storage costs grow forever.

Compute costs spike during high activity. Bull markets increase transaction volume. NFT mints create load spikes. Your infrastructure must handle peaks without overspending during quiet periods.

Network costs accumulate from data transfer. Syncing nodes downloads enormous amounts of data. API responses transfer to users. WebSocket connections consume bandwidth.

Here is a cost breakdown for a typical blockchain application:

RPC Providers: 40-60 percent of costs for most applications
Database Storage: 15-25 percent for indexed data

Compute: 15-20 percent for API servers and workers
Cache: 5-10 percent for Redis instances
Network: 5-10 percent for data transfer

SECTION 2: RPC COST OPTIMIZATION

RPC costs are usually the largest expense. Optimize ruthlessly.

```
from typing import Dict, List, Any, Optional
from dataclasses import dataclass
from collections import defaultdict
import time
import asyncio
```

```
@dataclass
class RPCCost:
    method: str
    compute_units: int
    archive_multiplier: float = 1.0
```

```
RPC_COSTS = {
    "eth_blockNumber": RPCCost("eth_blockNumber", 1),
    "eth_chainId": RPCCost("eth_chainId", 1),
    "eth_gasPrice": RPCCost("eth_gasPrice", 1),
    "net_version": RPCCost("net_version", 1),

    "eth_getBalance": RPCCost("eth_getBalance", 5,
archive_multiplier=2.0),
    "eth_getCode": RPCCost("eth_getCode", 5, archive_multiplier=2.0),
    "eth_getTransactionCount": RPCCost("eth_getTransactionCount", 5),

    "eth_call": RPCCost("eth_call", 10, archive_multiplier=3.0),
    "eth_estimateGas": RPCCost("eth_estimateGas", 15),

    "eth_getBlockByNumber": RPCCost("eth_getBlockByNumber", 10),
    "eth_getBlockByHash": RPCCost("eth_getBlockByHash", 10),
    "eth_getTransactionByHash": RPCCost("eth_getTransactionByHash",
5),
    "eth_getTransactionReceipt": RPCCost("eth_getTransactionReceipt",
```

```

10),

"eth_getLogs": RPCCost("eth_getLogs", 20),

"eth_sendRawTransaction": RPCCost("eth_sendRawTransaction",
50),

"debug_traceTransaction": RPCCost("debug_traceTransaction", 500),
"trace_call": RPCCost("trace_call", 100),
}

```

```

class RPCCostTracker:
    def __init__(self):
        self.usage: Dict[str, Dict[str, int]] = defaultdict(lambda:
defaultdict(int))
        self.daily_costs: Dict[str, int] = defaultdict(int)

    def record_call(self, method: str, is_archive: bool = False):
        cost_info = RPC_COSTS.get(method, RPCCost(method, 10))

        units = cost_info.compute_units
        if is_archive:
            units = int(units * cost_info.archive_multiplier)

        today = time.strftime("%Y-%m-%d")
        hour = time.strftime("%H")

        self.usage[today][method] += 1
        self.daily_costs[today] += units

    def get_daily_report(self, date: str = None) -> Dict:
        date = date or time.strftime("%Y-%m-%d")

        methods = self.usage.get(date, {})
        total_units = self.daily_costs.get(date, 0)

        sorted_methods = sorted(
            methods.items(),
            key=lambda x: x[1],

```

```
reverse = True
)
```

```
return {
    "date": date,
    "total_compute_units": total_units,
    "method_counts": dict(sorted_methods),
    "estimated_cost_usd": self._estimate_cost(total_units)
}
```

```
def _estimate_cost(self, units: int, price_per_million: float = 1.50) -> float:
    return (units / 1_000_000) * price_per_million
```

```
def get_optimization_suggestions(self) -> List[str]:
    suggestions = []
    today = time.strftime("%Y-%m-%d")
    methods = self.usage.get(today, {})
```

```
    if methods.get("eth_getLogs", 0) > 1000:
        suggestions.append(
            "High eth_getLogs usage. Consider batching log queries with larger block ranges."
        )
```

```
    if methods.get("eth_call", 0) > 5000:
        suggestions.append(
            "High eth_call usage. Implement multicall batching for contract reads."
        )
```

```
    balance_calls = methods.get("eth_getBalance", 0)
    if balance_calls > 1000:
        suggestions.append(
            f"Called eth_getBalance {balance_calls} times. Cache balances with short TTL."
        )
```

```
    if methods.get("debug_traceTransaction", 0) > 0:
        suggestions.append(
```

```
"debug_traceTransaction is expensive. Use only when necessary."  
)
```

```
return suggestions
```

```
class BatchOptimizer:
```

```
def __init__(self, rpc_client, max_batch_size: int = 100):
```

```
    self.rpc = rpc_client
```

```
    self.max_batch_size = max_batch_size
```

```
    async def batch_get_balances(  
        self,  
        addresses: List[str],  
        block: str = "latest"  
    ) -> Dict[str, int]:
```

```
        calls = [  
            {"method": "eth_getBalance", "params": [addr, block]}  
            for addr in addresses  
        ]
```

```
        results = {}
```

```
        for i in range(0, len(calls), self.max_batch_size):  
            batch = calls[i:i + self.max_batch_size]  
            batch_results = await self.rpc.batch_call(batch)
```

```
            for j, result in enumerate(batch_results):  
                addr = addresses[i + j]  
                results[addr] = int(result, 16) if result else 0
```

```
        return results
```

```
    async def batch_get_logs(  
        self,  
        contract_address: str,  
        from_block: int,  
        to_block: int,  
        batch_size: int = 2000
```

```
    ) -> List[Dict[str, Any]]:
```

```
        """Get logs for a contract address in a batch.  
        Returns a list of log dictionaries.  
        """
```

```
        logs = []  
        for i in range(0, len(calls), self.max_batch_size):  
            batch = calls[i:i + self.max_batch_size]  
            batch_results = await self.rpc.batch_call(batch)
```

```
            for j, result in enumerate(batch_results):  
                addr = addresses[i + j]  
                results[addr] = int(result, 16) if result else 0
```

```
        return results
```

```
    async def batch_get_logs(  
        self,  
        contract_address: str,  
        from_block: int,  
        to_block: int,  
        batch_size: int = 2000
```

```
    ) -> List[Dict[str, Any]]:
```

```
        """Get logs for a contract address in a batch.  
        Returns a list of log dictionaries.  
        """
```

```
        logs = []  
        for i in range(0, len(calls), self.max_batch_size):  
            batch = calls[i:i + self.max_batch_size]  
            batch_results = await self.rpc.batch_call(batch)
```

```
            for j, result in enumerate(batch_results):  
                addr = addresses[i + j]  
                results[addr] = int(result, 16) if result else 0
```

```
        return results
```

```
    async def batch_get_logs(  
        self,  
        contract_address: str,  
        from_block: int,  
        to_block: int,  
        batch_size: int = 2000
```

```
    ) -> List[Dict[str, Any]]:
```

```
        """Get logs for a contract address in a batch.  
        Returns a list of log dictionaries.  
        """
```

```
        logs = []  
        for i in range(0, len(calls), self.max_batch_size):  
            batch = calls[i:i + self.max_batch_size]  
            batch_results = await self.rpc.batch_call(batch)
```

```
            for j, result in enumerate(batch_results):  
                addr = addresses[i + j]  
                results[addr] = int(result, 16) if result else 0
```

```
        return results
```

```
    async def batch_get_logs(  
        self,  
        contract_address: str,  
        from_block: int,  
        to_block: int,  
        batch_size: int = 2000
```

```
    ) -> List[Dict[str, Any]]:
```

```
        """Get logs for a contract address in a batch.  
        Returns a list of log dictionaries.  
        """
```

```
        logs = []  
        for i in range(0, len(calls), self.max_batch_size):  
            batch = calls[i:i + self.max_batch_size]  
            batch_results = await self.rpc.batch_call(batch)
```

```
            for j, result in enumerate(batch_results):  
                addr = addresses[i + j]  
                results[addr] = int(result, 16) if result else 0
```

```
        return results
```

```
) -> List[Dict]:
```

```
all_logs = []
```

```
for start in range(from_block, to_block + 1, batch_size):  
    end = min(start + batch_size - 1, to_block)
```

```
logs = await self.rpc.call("eth_getLogs", [{  
    "address": contract_address,  
    "fromBlock": hex(start),  
    "toBlock": hex(end)  
}])
```

```
all_logs.extend(logs)
```

```
return all_logs
```

```
class MulticallOptimizer:  
    MULTICALL_ADDRESS =  
    "0xA11bde05977b3631167028862bE2a173976CA11"
```

```
    AGGREGATE_SIGNATURE = "0x252dba42"
```

```
    def __init__(self, rpc_client):  
        self.rpc = rpc_client
```

```
    async def multicall(  
        self,  
        calls: List[Dict[str, str]]  
    ) -> List[Any]:
```

```
        encoded_calls = []
```

```
        for call in calls:  
            encoded_calls.append((  
                call["target"],  
                call["callData"]  
            ))
```

```

from eth_abi import encode

call_data = self.AGGREGATE_SIGNATURE + encode(
    ["(address,bytes)[]"],
    [encoded_calls]
).hex()

result = await self.rpc.call("eth_call", [{
    "to": self.MULTICALL_ADDRESS,
    "data": call_data
}], "latest"])

from eth_abi import decode
decoded = decode(
    ["uint256", "bytes[]"],
    bytes.fromhex(result[2:])
)

return decoded[1]

async def batch_token_balances(
    self,
    token_address: str,
    addresses: List[str]
) -> Dict[str, int]:

    balance_selector = "0x70a08231"

    calls = []
    for addr in addresses:
        padded_addr = addr[2:].lower().zfill(64)
        calls.append({
            "target": token_address,
            "callData": balance_selector + padded_addr
        })

    results = await self.mutlicall(calls)

    balances = {}
    for i, result in enumerate(results):

```

```
balance = int.from_bytes(result, "big")
balances[addresses[i]] = balance

return balances
```

Batching and multicall reduce RPC costs by combining multiple queries into single requests.

SECTION 3: CACHING FOR COST REDUCTION

Caching is the most effective cost reduction strategy. Every cache hit is an RPC call saved.

```
class CostAwareCacheStrategy:
    def __init__(
        self,
        tiered_cache,
        rpc_cost_tracker: RPCCostTracker
    ):
        self.cache = tiered_cache
        self.cost_tracker = rpc_cost_tracker
        self.cache_hits = 0
        self.cache_misses = 0

    def calculate_savings(self) -> Dict:
        total_requests = self.cache_hits + self.cache_misses
        if total_requests == 0:
            return {"savings_percent": 0, "estimated_savings_usd": 0}

        hit_rate = self.cache_hits / total_requests

        avoided_units = self.cache_hits * 10
        savings_usd = self.cost_tracker.estimate_cost(avoided_units)

        return {
            "total_requests": total_requests,
            "cache_hits": self.cache_hits,
            "cache_misses": self.cache_misses,
            "hit_rate": hit_rate,
```

```
"avoided_compute_units": avoided_units,
"estimated_savings_usd": savings_usd
}
```

```
def get_optimal_ttl(self, data_type: str) -> int:
    ttl_recommendations = {
        "block_data": 0,
        "historical_transaction": 0,
        "confirmed_receipt": 0,

        "current_balance": 15,
        "nft_owner": 30,
        "token_price": 60,
        "gas_price": 10,

        "contract_metadata": 3600,
        "nft_metadata": 86400,
        "token_info": 3600,
        "ens_resolution": 300,

        "holder_list": 300,
        "floor_price": 60,
        "collection_stats": 120
    }
```

```
return ttl_recommendations.get(data_type, 60)
```

```
class IntelligentPrefetcher:
    def __init__(self, cache, rpc_client):
        self.cache = cache
        self.rpc = rpc_client
        self.access_patterns: Dict[str, List[float]] = defaultdict(list)
        self.prefetch_candidates: Dict[str, float] = {}

    def record_access(self, key: str):
        now = time.time()
        self.access_patterns[key].append(now)

    self.access_patterns[key] = [
```



```
t for t in self.access_patterns[key]
if now - t < 3600
]
```

```
if len(self.access_patterns[key]) >= 3:
self.prefetch_candidates[key] = now
```

```
async def prefetch_hot_keys(self):
now = time.time()
```

```
hot_keys = [
key for key, last_access in self.prefetch_candidates.items()
if now - last_access < 300
]
```

```
for key in hot_keys:
cached = await self.cache.get(key)
if cached is None:
await self._fetch_and_cache(key)
```

```
async def _fetch_and_cache(self, key: str):
pass
```

```
class CacheWarmingScheduler:
def __init__(self, cache, rpc_client, database):
self.cache = cache
self.rpc = rpc_client
self.db = database
```

```
async def warm_frequently_accessed(self):
top_addresses = await self.db.fetch(
"""
```

```
SELECT address, access_count
FROM address_access_log
WHERE accessed_at > NOW() - INTERVAL '24 hours'
GROUP BY address
ORDER BY access_count DESC
LIMIT 1000
""")
```

)

```
for row in top_addresses:
    await self.cache.get_balance(row["address"])
```

```
async def warm_active_collections(self):
    active_collections = await self.db.fetch(
        """
```

```
SELECT contract_address
FROM nft_transfers
WHERE block_timestamp > NOW() - INTERVAL '1 hour'
GROUP BY contract_address
ORDER BY COUNT(*) DESC
LIMIT 100
        """
```

)

```
for row in active_collections:
    pass
```

```
async def run_warming_schedule(self):
    while True:
        await self.warm_frequently_accessed()
        await asyncio.sleep(300)
```

Strategic caching and prefetching maximize cache hit rates to minimize RPC costs.

SECTION 4: DATABASE COST OPTIMIZATION

Database costs grow with data volume. Optimize storage and queries.

```
class DatabaseCostOptimizer:
    def __init__(self, database):
        self.db = database
```

```
    async def analyze_table_sizes(self) -> Dict:
        rows = await self.db.fetch(
```

```

"""
SELECT
schemaname,
tablename,
pg_size_pretty(pg_total_relation_size(schemaname || '.' ||
tablename)) as total_size,
pg_total_relation_size(schemaname || '.' || tablename) as size_bytes,
pg_size_pretty(pg_indexes_size(schemaname || '.' || tablename)) as
index_size
FROM pg_tables
WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(schemaname || '.' || tablename)
DESC
"""
)

```

```
total_bytes = sum(row["size_bytes"] for row in rows)
```

```

return {
    "tables": [dict(row) for row in rows],
    "total_size_bytes": total_bytes,
    "total_size_pretty": self._format_bytes(total_bytes)
}

```

```

def _format_bytes(self, bytes_val: int) -> str:
    for unit in ["B", "KB", "MB", "GB", "TB"]:
        if bytes_val < 1024:
            return f"{bytes_val:.2f} {unit}"
        bytes_val /= 1024
    return f"{bytes_val:.2f} PB"

```

```

async def identify_unused_indexes(self) -> List[Dict]:
    rows = await self.db.fetch(
        """

```

```

SELECT
schemaname,
tablename,
indexname,
pg_size_pretty(pg_relation_size(indexrelid)) as index_size,
idx_scan as scans

```

```

FROM pg_stat_user_indexes
WHERE idx_scan = 0
AND schemaname = 'public'
ORDER BY pg_relation_size(indexrelid) DESC
"""
)

```

```

return [dict(row) for row in rows]

```

```

async def get_slow_queries(self) -> List[Dict]:
rows = await self.db.fetch(
"""

```

```

SELECT
query,
calls,
total_time / 1000 as total_seconds,
mean_time as avg_ms,
rows
FROM pg_stat_statements
WHERE mean_time > 100
ORDER BY total_time DESC
LIMIT 20
"""
)

```

```

return [dict(row) for row in rows]

```

```

async def suggest_optimizations(self) -> List[str]:
suggestions = []

```

```

table_sizes = await self.analyze_table_sizes()
for table in table_sizes["tables"]:
if table["size_bytes"] > 10 * 1024 * 1024 * 1024:
suggestions.append(
f"Table {table['tablename']} is {table['total_size']}. "
f"Consider partitioning by block_number or timestamp."
)

```

```

unused_indexes = await self.identify_unused_indexes()
for idx in unused_indexes[:5]:

```

```

suggestions.append(
    f'Index {idx['indexname']} on {idx['tablename']} has 0 scans. '
    f'Consider dropping to save {idx['index_size']}.'
)

slow_queries = await self.get_slow_queries()
for query in slow_queries[:3]:
    suggestions.append(
        f'Query averaging {query['avg_ms']:.0f}ms. Consider optimization.'
    )

return suggestions

```

```

class DataRetentionManager:
    def __init__(self, database):
        self.db = database

    async def setup_partitioning(self, table_name: str):
        await self.db.execute(f"""
ALTER TABLE {table_name} RENAME TO {table_name}_old;

CREATE TABLE {table_name} (
LIKE {table_name}_old INCLUDING ALL
) PARTITION BY RANGE (block_number);

CREATE TABLE {table_name}_current PARTITION OF {table_name}
FOR VALUES FROM (18000000) TO (MAXVALUE);

INSERT INTO {table_name} SELECT * FROM {table_name}_old
WHERE block_number >= 18000000;
""")

    async def archive_old_data(
        self,
        table_name: str,
        retention_blocks: int,
        current_block: int
    ):
        cutoff_block = current_block - retention_blocks

```

```
archive_table = f'{table_name}_archive'
```

```
await self.db.execute(f"""  
CREATE TABLE IF NOT EXISTS {archive_table} (  
LIKE {table_name} INCLUDING ALL  
);
```

```
INSERT INTO {archive_table}  
SELECT * FROM {table_name}  
WHERE block_number < $1  
ON CONFLICT DO NOTHING;
```

```
DELETE FROM {table_name}  
WHERE block_number < $1;  
""", cutoff_block)
```

```
async def compress_old_partitions(self, table_name: str):  
    pass
```

```
async def delete_old_cache_entries(  
    self,  
    table_name: str,  
    max_age_days: int  
):  
    await self.db.execute(f"""  
DELETE FROM {table_name}  
WHERE created_at < NOW() - INTERVAL '{max_age_days} days'  
""")
```

```
class QueryOptimizer:  
    def __init__(self, database):  
        self.db = database
```

```
async def analyze_query(self, query: str) -> Dict:  
    explain = await self.db.fetch(  
        f'EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON) {query}'  
    )
```

```

plan = explain[0]["QUERY PLAN"][0]

return {
    "total_time_ms": plan.get("Execution Time", 0),
    "planning_time_ms": plan.get("Planning Time", 0),
    "plan": plan.get("Plan", {})
}

```

```

def suggest_indexes(self, query: str) -> List[str]:
    suggestions = []

```

```

    if "WHERE" in query.upper():
        pass

```

```

    return suggestions

```

Database optimization through partitioning, archiving, and index management reduces storage costs.

SECTION 5: COMPUTE RIGHT-SIZING

Match compute resources to actual workload.

```

import statistics
from typing import List, Dict
from datetime import datetime, timedelta

```

```

class ComputeAnalyzer:
    def __init__(self, metrics_client):
        self.metrics = metrics_client

    async def get_resource_utilization(
        self,
        service: str,
        period_hours: int = 24
    ) -> Dict:

```

```

        end_time = datetime.utcnow()
        start_time = end_time - timedelta(hours=period_hours)

```

```

cpu_usage = await self.metrics.query_range(
    f'rate(container_cpu_usage_seconds_total{{service="{service}"}}
[5m])',
    start_time,
    end_time
)

```

```

memory_usage = await self.metrics.query_range(
    f'container_memory_usage_bytes{{service="{service}"}}',
    start_time,
    end_time
)

```

```

cpu_values = [float(v) for v in cpu_usage]
memory_values = [float(v) for v in memory_usage]

```

```

return {
    "cpu": {
        "avg": statistics.mean(cpu_values) if cpu_values else 0,
        "max": max(cpu_values) if cpu_values else 0,
        "p95": self._percentile(cpu_values, 95) if cpu_values else 0
    },
    "memory": {
        "avg": statistics.mean(memory_values) if memory_values else 0,
        "max": max(memory_values) if memory_values else 0,
        "p95": self._percentile(memory_values, 95) if memory_values else 0
    }
}

```

```

def _percentile(self, data: List[float], percentile: int) -> float:
    if not data:
        return 0
    sorted_data = sorted(data)
    index = int(len(sorted_data) * percentile / 100)
    return sorted_data[min(index, len(sorted_data) - 1)]

```

```

def recommend_instance_size(self, utilization: Dict) -> Dict:
    cpu_p95 = utilization["cpu"]["p95"]
    memory_p95 = utilization["memory"]["p95"]

```



```

if cpu_p95 < 0.3 and memory_p95 < 0.3:
    recommendation = "downsize"
    reason = "Resources significantly underutilized"
elif cpu_p95 > 0.8 or memory_p95 > 0.8:
    recommendation = "upscale"
    reason = "Resources near capacity"
else:
    recommendation = "maintain"
    reason = "Resource utilization is appropriate"

return {
    "recommendation": recommendation,
    "reason": reason,
    "cpu_utilization": cpu_p95,
    "memory_utilization": memory_p95
}

```

```

class AutoScalingOptimizer:
    def __init__(self):
        self.scaling_events: List[Dict] = []

    def analyze_scaling_efficiency(self) -> Dict:
        if not self.scaling_events:
            return {"efficiency": "unknown"}

        scale_ups = [e for e in self.scaling_events if e["direction"] == "up"]
        scale_downs = [e for e in self.scaling_events if e["direction"] == "down"]

        avg_scale_up_time = statistics.mean(
            [e["duration_seconds"] for e in scale_ups]
        ) if scale_ups else 0

        return {
            "total_scaling_events": len(self.scaling_events),
            "scale_ups": len(scale_ups),
            "scale_downs": len(scale_downs),

```

```
"avg_scale_up_time_seconds": avg_scale_up_time
}
```

```
def recommend_scaling_policy(
self,
utilization: Dict,
traffic_patterns: Dict
) -> Dict:
```

```
recommendations = {}
```

```
if traffic_patterns.get("predictable"):
recommendations["type"] = "scheduled"
recommendations["schedule"] =
self._generate_schedule(traffic_patterns)
else:
recommendations["type"] = "metric_based"
recommendations["cpu_threshold"] = 70
recommendations["memory_threshold"] = 75
recommendations["scale_up_cooldown"] = 180
recommendations["scale_down_cooldown"] = 300
```

```
return recommendations
```

```
def _generate_schedule(self, patterns: Dict) -> List[Dict]:
return [
{"cron": "0 8 * * 1-5", "replicas": 5},
{"cron": "0 18 * * 1-5", "replicas": 2},
{"cron": "0 0 * * 6-7", "replicas": 1}
]
```

```
class SpotInstanceManager:
def __init__(self):
self.spot_savings = 0
self.interruptions = 0
```

```
def calculate_spot_savings(
self,
on_demand_hours: int,
```

```
on_demand_price: float,  
spot_hours: int,  
spot_price: float  
) -> Dict:
```

```
on_demand_cost = on_demand_hours * on_demand_price  
spot_cost = spot_hours * spot_price  
savings = on_demand_cost - spot_cost
```

```
return {  
    "on_demand_cost": on_demand_cost,  
    "spot_cost": spot_cost,  
    "savings": savings,  
    "savings_percent": (savings / on_demand_cost) * 100 if  
on_demand_cost > 0 else 0  
}
```

```
def recommend_spot_usage(self, workload_type: str) -> Dict:  
recommendations = {  
    "indexer": {  
        "spot_suitable": True,  
        "reason": "Indexing can resume from checkpoint after interruption",  
        "recommended_spot_percent": 80  
    },  
    "api_server": {  
        "spot_suitable": True,  
        "reason": "Stateless and behind load balancer",  
        "recommended_spot_percent": 50  
    },  
    "transaction_sender": {  
        "spot_suitable": False,  
        "reason": "Transaction nonce state could be corrupted",  
        "recommended_spot_percent": 0  
    },  
    "database": {  
        "spot_suitable": False,  
        "reason": "Data durability requirements",  
        "recommended_spot_percent": 0  
    }  
}
```

```

return recommendations.get(workload_type, {
    "spot_suitable": False,
    "reason": "Unknown workload type",
    "recommended_spot_percent": 0
})

```

Right-sizing and spot instances significantly reduce compute costs.

SECTION 6: NETWORK COST OPTIMIZATION

Network transfer costs accumulate, especially across regions.

```

class NetworkCostOptimizer:
    def __init__(self):
        self.transfer_log: List[Dict] = []

    def estimate_node_sync_cost(
        self,
        chain: str,
        sync_type: str
    ) -> Dict:

        sync_data = {
            "ethereum": {
                "full": {"gb": 800, "time_hours": 48},
                "snap": {"gb": 400, "time_hours": 8},
                "archive": {"gb": 15000, "time_hours": 168}
            },
            "polygon": {
                "full": {"gb": 2000, "time_hours": 72},
                "archive": {"gb": 8000, "time_hours": 168}
            }
        }

        chain_data = sync_data.get(chain, {})
        sync_info = chain_data.get(sync_type, {"gb": 500, "time_hours":
24})

```

```
cost_per_gb = 0.09
total_cost = sync_info["gb"] * cost_per_gb
```

```
return {
    "chain": chain,
    "sync_type": sync_type,
    "data_gb": sync_info["gb"],
    "time_hours": sync_info["time_hours"],
    "estimated_transfer_cost": total_cost
}
```

```
def optimize_response_size(self, response: Dict) -> Dict:
    if "transactions" in response:
        response["transactions"] = [
            self._minimize_transaction(tx)
            for tx in response["transactions"]
        ]
```

```
    return response
```

```
def _minimize_transaction(self, tx: Dict) -> Dict:
    essential_fields = ["hash", "from", "to", "value", "blockNumber"]
    return {k: v for k, v in tx.items() if k in essential_fields}
```

```
def recommend_compression(self, content_type: str) -> Dict:
    recommendations = {
        "json": {
            "algorithm": "gzip",
            "typical_ratio": 0.15,
            "cpu_overhead": "low"
        },
        "binary": {
            "algorithm": "zstd",
            "typical_ratio": 0.25,
            "cpu_overhead": "medium"
        }
    }
```

```
    return recommendations.get(content_type,
        recommendations["json"])
```

```
def calculate_cdn_savings(
    self,
    requests_per_month: int,
    avg_response_kb: float,
    cache_hit_rate: float
) -> Dict:
```

```
total_data_gb = (requests_per_month * avg_response_kb) / (1024 *
1024)
```

```
origin_cost_per_gb = 0.09
cdn_cost_per_gb = 0.02
```

```
without_cdn = total_data_gb * origin_cost_per_gb
with_cdn = (
    total_data_gb * (1 - cache_hit_rate) * origin_cost_per_gb +
    total_data_gb * cache_hit_rate * cdn_cost_per_gb
)
```

```
return {
    "monthly_requests": requests_per_month,
    "total_data_gb": total_data_gb,
    "cost_without_cdn": without_cdn,
    "cost_with_cdn": with_cdn,
    "savings": without_cdn - with_cdn
}
```

Compression, CDN caching, and minimized responses reduce network transfer costs.

SECTION 7: PROVIDER COST COMPARISON

Different providers have different pricing. Choose wisely.

```
@dataclass
class ProviderPricing:
    name: str
    base_monthly: float
```

included_compute_units: int
overage_per_million: float
archive_multiplier: float
websocket_included: bool

```
PROVIDER_PRICING = [  
    ProviderPricing(  
        name="Alchemy Growth",  
        base_monthly=49,  
        included_compute_units=40_000_000,  
        overage_per_million=1.20,  
        archive_multiplier=1.5,  
        websocket_included=True  
    ),  
    ProviderPricing(  
        name="Infura Core",  
        base_monthly=50,  
        included_compute_units=100_000,  
        overage_per_million=0.0,  
        archive_multiplier=2.0,  
        websocket_included=True  
    ),  
    ProviderPricing(  
        name="QuickNode Build",  
        base_monthly=49,  
        included_compute_units=30_000_000,  
        overage_per_million=1.50,  
        archive_multiplier=1.0,  
        websocket_included=True  
    ),  
    ProviderPricing(  
        name="Self-hosted Full Node",  
        base_monthly=200,  
        included_compute_units=999_999_999,  
        overage_per_million=0.0,  
        archive_multiplier=1.0,  
        websocket_included=True  
    ),  
    ProviderPricing(  
        name="Self-hosted Archive",
```

```

base_monthly = 800,
included_compute_units = 999_999_999,
overage_per_million = 0.0,
archive_multiplier = 1.0,
websocket_included = True
)
]

```

```

class ProviderCostCalculator:
    def __init__(self):
        self.providers = {p.name: p for p in PROVIDER_PRICING}

    def calculate_monthly_cost(
        self,
        provider_name: str,
        compute_units: int,
        archive_percentage: float = 0
    ) -> Dict:

        provider = self.providers.get(provider_name)
        if not provider:
            return {"error": "Unknown provider"}

        regular_units = compute_units * (1 - archive_percentage)
        archive_units = compute_units * archive_percentage *
        provider.archive_multiplier

        total_units = regular_units + archive_units

        if total_units <= provider.included_compute_units:
            total_cost = provider.base_monthly
            overage_cost = 0
        else:
            overage_units = total_units - provider.included_compute_units
            overage_cost = (overage_units / 1_000_000) *
            provider.overage_per_million
            total_cost = provider.base_monthly + overage_cost

        return {

```



```

"provider": provider_name,
"base_monthly": provider.base_monthly,
"compute_units_used": compute_units,
"effective_units": total_units,
"included_units": provider.included_compute_units,
"overage_units": max(0, total_units -
provider.included_compute_units),
"overage_cost": overage_cost,
"total_monthly_cost": total_cost
}

```

```

def compare_providers(
self,
compute_units: int,
archive_percentage: float = 0
) -> List[Dict]:

```

```

comparisons = []

```

```

for provider in PROVIDER_PRICING:
cost = self.calculate_monthly_cost(
provider.name,
compute_units,
archive_percentage
)
comparisons.append(cost)

```

```

comparisons.sort(key = lambda x: x["total_monthly_cost"])

```

```

return comparisons

```

```

def find_breakeven_point(
self,
provider_a: str,
provider_b: str
) -> Optional[int]:

```

```

a = self.providers.get(provider_a)
b = self.providers.get(provider_b)

```

```
if not a or not b:  
    return None
```

```
for units in range(0, 500_000_000, 1_000_000):  
    cost_a = self.calculate_monthly_cost(provider_a, units)  
    ["total_monthly_cost"]  
    cost_b = self.calculate_monthly_cost(provider_b, units)  
    ["total_monthly_cost"]
```

```
if cost_a < cost_b:  
    return units
```

```
return None
```

Provider comparison helps choose the most cost-effective option for your usage pattern.

SECTION 8: COST MONITORING DASHBOARD

Track costs in real-time to prevent surprises.

```
class CostMonitoringDashboard:  
    def __init__(  
        self,  
        rpc_tracker: RPCCostTracker,  
        database_optimizer: DatabaseCostOptimizer,  
        compute_analyzer: ComputeAnalyzer  
    ):  
        self.rpc = rpc_tracker  
        self.db = database_optimizer  
        self.compute = compute_analyzer  
  
    async def get_daily_cost_summary(self) -> Dict:  
        rpc_report = self.rpc.get_daily_report()  
  
        db_sizes = await self.db.analyze_table_sizes()  
        db_cost = self.estimate_storage_cost(db_sizes["total_size_bytes"])  
  
        compute_cost = 50.0
```

```
network_cost = 20.0
```

```
total = (  
    rpc_report["estimated_cost_usd"] +  
    db_cost +  
    compute_cost +  
    network_cost  
)
```

```
return {  
    "date": rpc_report["date"],  
    "breakdown": {  
        "rpc": rpc_report["estimated_cost_usd"],  
        "database": db_cost,  
        "compute": compute_cost,  
        "network": network_cost  
    },  
    "total_daily_cost": total,  
    "projected_monthly_cost": total * 30  
}
```

```
def _estimate_storage_cost(self, bytes_val: int) -> float:  
    gb = bytes_val / (1024 * 1024 * 1024)  
    cost_per_gb_month = 0.115  
    daily_cost = (gb * cost_per_gb_month) / 30  
    return daily_cost
```

```
async def get_cost_trends(self, days: int = 30) -> List[Dict]:  
    trends = []
```

```
    for i in range(days):  
        date = (datetime.now() - timedelta(days=i)).strftime("%Y-%m-%d")  
        daily = self.rpc.daily_costs.get(date, 0)  
        trends.append({  
            "date": date,  
            "compute_units": daily,  
            "estimated_cost": self.rpc._estimate_cost(daily)  
        })
```

```
return trends
```

```
async def get_optimization_opportunities(self) -> List[Dict]:  
opportunities = []
```

```
rpc_suggestions = self.rpc.get_optimization_suggestions()  
for suggestion in rpc_suggestions:  
opportunities.append({  
"category": "rpc",  
"suggestion": suggestion,  
"potential_savings": "10-30%"  
})
```

```
db_suggestions = await self.db.suggest_optimizations()  
for suggestion in db_suggestions:  
opportunities.append({  
"category": "database",  
"suggestion": suggestion,  
"potential_savings": "5-20%"  
})
```

```
return opportunities
```

```
async def set_budget_alert(  
self,  
daily_budget: float,  
alert_threshold: float = 0.8  
):  
current = await self.get_daily_cost_summary()  
current_cost = current["total_daily_cost"]
```

```
if current_cost > daily_budget * alert_threshold:  
return {  
"alert": True,  
"message": f"Daily cost ${current_cost:.2f} exceeds  
{alert_threshold*100}% of budget ${daily_budget:.2f}",  
"current_cost": current_cost,  
"budget": daily_budget  
}
```

```

return {
    "alert": False,
    "current_cost": current_cost,
    "budget": daily_budget,
    "remaining": daily_budget - current_cost
}

```

Cost monitoring provides visibility into spending and identifies optimization opportunities.

SECTION 9: COST OPTIMIZATION CHECKLIST

Follow this checklist to minimize infrastructure costs.

```

class CostOptimizationChecklist:
    def __init__(self):
        self.checks = []

    async def run_audit(self, context: Dict) -> Dict:
        results = []

        rpc_checks = [
            ("Use batch requests for multiple queries",
             context.get("uses_batching", False)),
            ("Implement multicall for contract reads",
             context.get("uses_multicall", False)),
            ("Cache RPC responses appropriately", context.get("cache_hit_rate",
             0) > 0.7),
            ("Use WebSocket for subscriptions", context.get("uses_websocket",
             False)),
            ("Avoid archive queries when possible",
             context.get("archive_percentage", 1) < 0.1)
        ]

        for check, passed in rpc_checks:
            results.append({
                "category": "RPC",
                "check": check,

```

```
"passed": passed,  
"impact": "high"  
})
```

```
db_checks = [  
    ("Partition large tables", context.get("uses_partitioning", False)),  
    ("Archive old data", context.get("has_retention_policy", False)),  
    ("Remove unused indexes", context.get("unused_indexes", 1) ==  
0),  
    ("Use connection pooling", context.get("uses_connection_pool",  
False))  
]
```

for check, passed in db_checks:

```
results.append({  
    "category": "Database",  
    "check": check,  
    "passed": passed,  
    "impact": "medium"  
})
```

```
compute_checks = [  
    ("Right-size instances", context.get("cpu_utilization", 0) > 0.3),  
    ("Use auto-scaling", context.get("uses_autoscaling", False)),  
    ("Consider spot instances", context.get("uses_spot", False)),  
    ("Use reserved instances for stable workloads",  
context.get("uses_reserved", False))  
]
```

for check, passed in compute_checks:

```
results.append({  
    "category": "Compute",  
    "check": check,  
    "passed": passed,  
    "impact": "high"  
})
```

```
passed = sum(1 for r in results if r["passed"])  
total = len(results)
```

```

return {
    "score": passed / total * 100,
    "passed": passed,
    "total": total,
    "checks": results,
    "recommendations": [
        r["check"] for r in results if not r["passed"]
    ]
}

```

SECTION 10: COMPLETE COST OPTIMIZATION SYSTEM

Here is a complete cost optimization implementation.

```

class CostOptimizationSystem:
    def __init__(self, config: Dict):
        self.config = config
        self.rpc_tracker = RPCCostTracker()
        self.batch_optimizer = None
        self.multicall_optimizer = None
        self.db_optimizer = None
        self.compute_analyzer = None
        self.dashboard = None

    async def initialize(self, rpc_client, database, metrics_client):
        self.batch_optimizer = BatchOptimizer(rpc_client)
        self.multicall_optimizer = MulticallOptimizer(rpc_client)
        self.db_optimizer = DatabaseCostOptimizer(database)
        self.compute_analyzer = ComputeAnalyzer(metrics_client)

        self.dashboard = CostMonitoringDashboard(
            self.rpc_tracker,
            self.db_optimizer,
            self.compute_analyzer
        )

    async def get_batch_balances(
        self,
        addresses: List[str]

```

```
) -> Dict[str, int]:
```

```
for addr in addresses:
```

```
self.rpc_tracker.record_call("eth_getBalance")
```

```
return await self.batch_optimizer.batch_get_balances(addresses)
```

```
async def get_token_balances_multicall(
```

```
self,
```

```
token: str,
```

```
addresses: List[str]
```

```
) -> Dict[str, int]:
```

```
self.rpc_tracker.record_call("eth_call")
```

```
return await self.multicall_optimizer.batch_token_balances(token,  
addresses)
```

```
async def get_cost_report(self) -> Dict:
```

```
daily_summary = await self.dashboard.get_daily_cost_summary()
```

```
opportunities = await
```

```
self.dashboard.get_optimization_opportunities()
```

```
trends = await self.dashboard.get_cost_trends(7)
```

```
return {
```

```
"summary": daily_summary,
```

```
"optimization_opportunities": opportunities,
```

```
"trends": trends,
```

```
"rpc_breakdown": self.rpc_tracker.get_daily_report()
```

```
}
```

```
async def check_budget(self, daily_budget: float) -> Dict:
```

```
return await self.dashboard.set_budget_alert(daily_budget)
```

```
async def run_optimization_audit(self) -> Dict:
```

```
context = {
```

```
"uses_batching": True,
```

```
"uses_multicall": True,
```

```
"cache_hit_rate": 0.75,
```

```
"uses_websocket": True,
```



```
"archive_percentage": 0.05,
"uses_partitioning": True,
"has_retention_policy": True,
"unused_indexes": 0,
"uses_connection_pool": True,
"cpu_utilization": 0.5,
"uses_autoscaling": True,
"uses_spot": True,
"uses_reserved": False
}
```

```
checklist = CostOptimizationChecklist()
return await checklist.run_audit(context)
```

```
async def main():
    system = CostOptimizationSystem({})

    report = await system.get_cost_report()
    print(f'Daily Cost Summary: {report['summary']}')

    audit = await system.run_optimization_audit()
    print(f'Optimization Score: {audit['score']:.1f}%')
    print(f'Recommendations: {audit['recommendations']}')

    budget_status = await system.check_budget(100.0)
    print(f'Budget Status: {budget_status}')

if __name__ == "__main__":
    asyncio.run(main())
```

This chapter covered cost optimization for blockchain infrastructure. You learned RPC cost tracking and batching strategies, caching to reduce expensive RPC calls, database optimization through partitioning and archiving, compute right-sizing and spot instances, network optimization with compression and CDN, provider cost comparison, and cost monitoring dashboards. Infrastructure costs can easily exceed revenue without optimization. Apply these strategies to keep your blockchain application profitable.

This completes Book 4: Infrastructure and Scaling. You now have the knowledge to build production-grade blockchain infrastructure that is reliable, secure, and cost-effective.